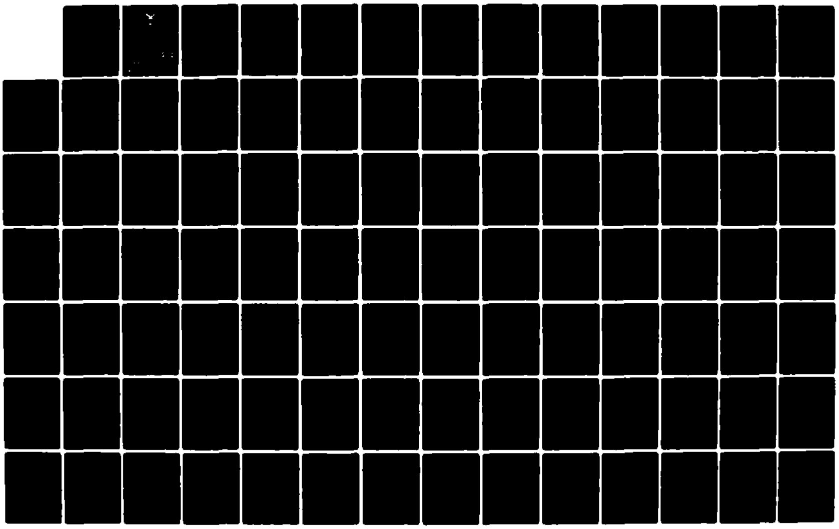
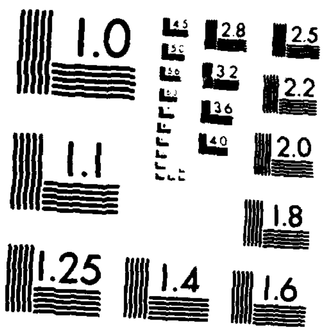




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A124694

AIR INSTITUTE OF TECHNOLOGY

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

[illegible]

83 02 [illegible]

AFIT/GE/EE/82D-58

DEVELOPMENT OF AN INTERACTIVE COMPUTER GRAPHICS SYSTEM LIBRARY AND GRAPHICS TOOLS

THESIS

AFIT/GE/EE/82D-58 Kevin W. Rose
CAPT USAF

AFIT/GE/EE/82D-58

DEVELOPMENT OF AN INTERACTIVE COMPUTER GRAPHICS SYSTEM LIBRARY AND GRAPHICS TOOLS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

in Partial Fullfillment of the

Requirements for the Degree of

Master of Science

by

Kevin W. Rose, B.S.

CAPT USAF

Graduate Electrical Engineering

December 1982

Preface

Two previous research efforts at AFIT laid a foundation on which to build a graphics capability for the VAX 11/780 in the Digital Engineering Laboratory at AFIT. Harold Curling provided a design for a set of input routines to be added to a standardized graphics output package. Phillip Tarbell investigated the availability of several implementations of standardized graphics packages that led to the acquisition of GWCORE, an implementation by George Washington University. This report describes the implementation and extension of GWCORE to provide an extensive graphics capability for the VAX computer and its graphics peripherals.

I would like to thank several individuals who were helpful during the course of this investigation. My thesis advisor, Dr. Gary Lamont, deserves thanks for providing guidance when needed and still allowing me the freedom to develop a variety of test and applications programs. Professor Charles Richard and Captain Roie Black were helpful with their comments and criticisms of this report. Mr. Bill McQuay of the Avionics Laboratory provided the much needed access to his VAX at the start and conclusion of this work.

Most of all I need to thank my wife, Linda Kay. Her work in commercial applications of computer graphics motivated my interest in graphics, and her companionship throughout this effort made it all worthwhile.

Contents

Contents

List of Figures

List of Tables

Abstract

The requirements for a general purpose, interactive graphics software system were investigated and led to the development of a graphics library of subroutines which can be used as a graphics research tool or for applications development. The design objectives for software environments and software tools established the strategy for developing the graphics software library. It was determined that the ACM Core standard graphics specification satisified many of the requirements. An output-only implementation of the Core by George Washington University was extended to include a device driver for a color display device and the synchronous input primitives. The graphics system was used to develop an application program to demonstrate the design of an effective user-tool interface. The program is an interactive graphics editor that allows preparation of two-dimensional pictures and includes capabilities for picture storage and retrieval, segment addition and deletion, attribute control, error handling and on-line help. The graphics software library implemented to the specification of the Core standard proposal provides an effective tool for interactive computer graphics education, research, and applications development.

DEVELOPMENT OF AN INTERACTIVE COMPUTER GRAPHICS SYSTEM LIBRARY AND GRAPHICS TOOLS

I. Introduction

The purpose of this investigation is to continue the development of software for use with a general-purpose interactive computer graphics system. The development of the interactive graphics software is to be guided by the methodologies of software engineering, particularly those methods directed toward producing software tools. It will be shown that the proposed graphics standard of the Association for Computing Machinery's Special Interest Group on Graphics (ACM SIGGRAPH) satisfies the requirements for software tools in general as well as specifying a highly capable graphics system.

The Air Force Avionics Laboratory, Air Force Materials Laboratory and the Air Force Institute of Technology (AFIT) make extensive use of computer graphics in a variety of disciplines. As sponsors for this effort, they have a requirement for producing a system of graphics subroutines and integrating them into existing programming environments to provide a tool for subsequent software development. The graphics software is to be usable with a variety of host processors and display devices and therefore should be as device independent as possible. The graphics package should be extensible to allow advanced graphics software to be incorporated. The graphics package should be modular to allow some or all of its capabilities to be incorporated in other application programs. The combination of device independence, extensibility, and modularity are characteristics desirable for all software tools [9]. The effort to be documented herein will determine the desirable traits of software tools and will incorporate them into the overall development of a graphics capability.

The remaining sections of this chapter discuss background material on computer graphics systems and software tools, a statement of the problem, the scope of this effort, past development efforts, and the development approach.

Development of Computer Graphics

Computer graphics provides a means of user interaction based on symbols or pictorial content that has a much higher information bearing content than textual material. Graphics is more useful than text for assimilating spatial designs, relationships and raw numerical data[31]. Early graphics, such as Sutherland's Sketchpad, allowed drawing on a cathode ray tube screen, positioning and moving standard component symbols, and allowed storing assembled hierarchies in a data structure [1:22]. Other graphics systems developed by General Motors and Boeing were used to display solid objects as an aid to the drawing intensive process of mechanical design[1:22]. Computer aided design and manufacturing remain significant areas of application today with nearly two-thirds of the graphics market devoted to sales of computer aided design systems[32:68]. The burgeoning areas of applications now include electronic design, industrial drawing, business administration, artistic rendering, robotics, command and control, image processing and architectural drawing.

The underlying methods of interactive computer graphics have been well defined and documented [for example, Ref 2]. The fundamental methods of line-drawing, user interaction, transformations, projections, clipping, and modelling are common to numerous applications. These techniques have become the subject of attempts at standardization [3].

The proposed standards have been carefully composed to maximize the graphics capability of any computing system with a graphics display. The emphasis of the standards has been on defining device independent graphics software. Device independence means that only at the lowest level of actually driving a display device is the software dependent on specific machine functions. The remaining software specification provides a sophisticated set of capabilities to supply high quality graphics production independent of the particular host processor and display devices.

Development of Software Tools

A brief history of the development of software tools is provided by Edward Miller in the introduction to "Tutorial: Automated Tools for Software Engineering" [4]. The development of software tools goes back to the early days of computing; among the first tools were linkers, loaders, and assemblers. As computers evolved, so did the tools for software production. Symbolic assemblers allowed the use of mnemonics as variable and subroutine names. Relative assemblers allowed relocating executable code in memory. The use of high level languages to improve programming efficiency and readibility required compilers. Interactive computing led to new tools such as line and screen oriented text editors for program development, and interactive debuggers for program testing.

With the advent of high level languages came efforts to structure the language to reduce the software system design and implementation difficulties. The rigorous structuring was applied to the earlier phases of software development: namely, analysis and design. The division of the software development process into life-cycle phases has been followed by development of tools to support specific functions of the life-cycle. Tools can be categorized by their support for requirements specification, design, implementation, testing, and maintenance. This view of the software development environment shows that tools are aids for designing, coding, testing, and executing software over the entire life cycle of a software system [7].

There exists a second level for a hierarchical description of software tools. The view of each software tool as a component to be used in a larger software system arises from the software engineering methodology that encourages breaking a large problem into several smaller problems. In this way, a comprehensive solution to a large problem is obtained by combining the results from the individual components. Similarly, each software tool provides a specialized operation but when used together in an environment they provide a collection of flexible utilities for many users.

When the tools are integrated into a whole, they constitute a particular software development environment. Recent efforts in environment development have been devoted to ascertaining the useful qualities of the environment and constituent tools [for example, Ref 5]. These qualities should prove useful as guidelines for developing a graphics capability and in forging useful graphics tools.

Problem Statement and Significance

Various computing facilities at Wright-Patterson Air Force Base (WPAFB) make extensive use of computer graphics. The implementation of the Core standard was proposed by the Avionics Laboratory Electronic Warfare Analysis Group and the Air Force Institute of Technology. Other interested offices include the Long Range Combat Aircraft Program Office of the Aeronautical Systems Division, and the computer support branch of the Materials Laboratory. The problems of graphics utilization for each of these groups is identical to the problem at large, that is, effectively using the programming talent available for production of graphics on a wide variety of equipment for an equally wide variety of applications. The objective of the computer graphics work at the Air Force Institute of Technology (AFIT) is to provide a flexible graphics facility to support research into graphics techniques, graphics applications and WPAFB laboratory research and development facilities.

These offices have common areas of application and similar host processors and displays. However, the offices also have unique display devices and unique applications. At present, any graphics software developed by one group is, in general, not usable by any other group due to device-specific requirements. The software development engineer is not "transportable" either, i.e. the software development personnel will require training on each different computer system, display device, and application at hand.

The motivation for establishing a device independent graphics tool is to provide standard graphics software for each installation. This standard software should allow applications software to be transferred between installations and be usable on all of the display devices at any given site without extensive modification. The application of design criteria from software engineering environments to the development of a computer graphics workbench would provide a useful tool to the applications programmer.

The significance of this capability can relate to a substantial improvement in productivity. One application can be used in more than one environment. Furthermore, the application developer, who must develop unique applications software, is presented with a capability at one site similar to any other site and is not forced to reinvent the fundamental graphics code and relearn a device specific graphics language. The resource manager for the computer system benefits from improved programmer productivity and better utilization of the systems capabilities.

Scope and Assumptions

There are two aspects of the investigation which will be limited. First, there exists a large number of software tools with a correspondingly large number of characteristics and applications. The review of programming environments in general and software tools in particular will result in ascertaining the qualities of environments and tools that provide the greatest value to development of a graphics tool. The review is not to provide a listing of all available tools and their possible applications to either software development or graphics. The implementations of the graphics tools are to be along the lines of developing graphics utilities useful within larger applications programs (rather than a software development tool to support the life-cycle of a graphics system.)

Second, there also exists a large number of graphics devices and software packages. Like the tools, only the software that satisfies the graphics system requirements will be reviewed and considered for further development. An explanation of graphics fundamentals will not duplicate material more extensively presented in the body of literature on graphics [for example 1, 2, 3, 24, 25, 34, 35].

Past Development Efforts

The most significant effort to date has been the ACM Core standard proposal for computer graphics software. Extensive literature exists on the desirabilty of a standard specification for graphics capabilities and the design issues [25, 34]. A survey of implementations has also been compiled [6]. Previous efforts at AFIT provide a foundation for implementation of the Core standard on host computer systems available at AFIT. The initial work [7] produced a design of the input primitives for inclusion in an output-only version of the Core standard. Another research effor [8] led to the acquisition of two implementations of the

Core standard [30,36] for use with the AFIT Digital Equipment Corporation VAX 11/780.

Approach

The requirements for the functional capabilities of a graphics system will be defined first. The emphasis will be on the requirements for software capabilites to support the current equipment but will not preclude additional devices that may be added at a later date. The other aspect of the requirements for the graphics system will come from the literature on programming environments and software tools. These additional requirements will be applied to the overall system design and particularly the implementation of the applications programs.

Following the requirements definition, the design options are developed. The advantages and disadvantages of the implementation alternatives will lead to selection of an approach for development and extension of the graphics software. The existing implementations will be assessed for availability, completeness, and suitability for integration into the AFIT VAX environment.

Finally, the utility of the implementation will be tested by developing a suitable application of the graphics software. The implementation of the graphics capabilities and the development of application programs will be in consonance with the objectives of good tool design. The goals of this investigation are to ascertain the qualities of good graphics software design and apply them to produce effective components of the graphics environment.

Overview of Thesis

The next chapter discusses the origin of software development environments and examines the desirable qualities that pertain to environments in general. Following this the requirements for a graphics system are presented. The third chapter specifies the hardware used for this investigation, examines the software options and presents a rationale for selection of a particular design. The third chapter concludes by relating testing and development objectives to the requirements. The fourth chapter presents the details of the extensions to the existing capabilities and implementation of the graphics tools. This includes implementation of the input routines, device drivers, and design and implementation of the applications programs. Chapter 5 presents the results of the implementation, identifies the requirements for additional work and offers some observations on the implementation of standard software routines and applications development.

II. Requirements Definition

It is from the perspective of developing a graphics environment that the requirements definition for a graphics system will proceed. This chapter presents a discussion of programming environments and the environments' components, software tools. The motivation for examining the design strategies of programming environments and tools is presented by way of example. Examining the implementation, structure, and interaction of three environments will prove useful for the development of a graphics system design strategy.

For the graphics system itself, both the hardware and software requirements of a general purpose graphics system are presented. Although no hardware is to be designed for this investigation, the graphics hardware imposes requirements on the software to be developed. The chapter concludes with a statement of the requirements for a graphics environment as derived from programming environments and graphics system development.

Programming Environments and Software Tools

The genesis of programming environments is a result of the need to improve the productivity of the software engineer. The rising cost of software development and falling cost of the hardware emphasizes the need for improving the quality of software. Historically, the environment was not much more than an editor, a compiler and perhaps a debugger [4]. Programming emphasis was on coding flowcharts into a suitable language. When software development was consistently over budget, late and the software did not fulfill the users' needs, it was apparent that the development process needed examination. A structured design approach has evolved and the subsequent application of software engineering practices for structured design and development improved the production process. The introduction of structured methodologies applied over the software life-cycle led to the development of automated software tools that aided in requirements definition, structured design, structured programming, testing and instrumentation, validation, and maintenance. The environments supported some or all of these phases of software development. The form a specific environment assumed was a result of the needs of the particular production process.

The requirements of the users and developers have given rise to two general forms of the environments [14]. One form is a collection of tools and utilities that support a variety of development requirements. The tools may be redundant and no particular programming language or development methodology guides the application of the tools. The most widely used environment of this form consists of the UNIX operating system and shell and the tools of the Programmer's Work Bench [10].

A second form for the environment is the result of integrating the tools to support a particular development methodology or programming language. In this case, the tools are not redundant, and the output of one phase of the development life-cycle is input to the next tool for the next phase. The application of the tool set is supported by an underlying method of development. The method requires systematic handling of the developmert from initial requirements definition through software maintenance. One of the most recently developed integrated environments is for software development using the Department of Defense programming language Ada. The environment is designed to support development of programs in one language and the tools are useful only with Ada program development.

The general software development environment is defined by the available tools. Kernighan and Plauger define a tool as that which solves a general problem and is so easy to use that people use it instead of building their own [9]. This definition of a tool results from their view of a tool as a software module designed to perform a single function. Tools allow applications to be assembled from existing programs rather than designed and built from scratch. Tools work together for a cumulative effect greater than similar programs not easily connected.

Contemporary environments include the kernel tools of editors, compilers, and debuggers, the lower level tools that provide file handling, formatting of text, and the tools to support the development life-cycle such as the requirements and design tools, automated source code control, validation tools and a data base that automatically keeps track of the time spent by programmers during development, updates schedules, and maintains configuration control [5:6].

Tools assist the developer of a software system by supporting the development process. Computer graphics tools should support the development of additional graphics software as well as providing the means of production for the final image. The use of graphics as an integral part of the environment is, at present, only at the research stage. Zelkowitz states that the input/output device that connects the user to an environment will greatly affect programmer productivity [5:42]. Most current terminal devices, even with a CRT, are based on the basic teletypewriter where the user types in lines of text and the device responds with lines of text. Graphics will have an increasingly important role in the development of future devices. High resolution and color display properties add new dimensions to the user-machine interface that have not been fully explored. The use of graphics in software development environments varies from none in the Ada environment [1:178] to complete reliance on graphics in the Smalltalk environment [13]. It is useful to examine some of these widely accepted and recently developed environments to establish the overall qualities for a successful implementation.

The UNIX Environment

The UNIX environment has proven its utility to support software production by its rapid acceptance at a large number of installations. Beginning in 1969 at Bell Telephone Laboratories, Ken Thompson and Dennis Ritchie developed the UNIX operating system from the outset to support interactive processes [21]. The addition of a collection of editors, text formatters, high-level-language compilers and debugging aids provided the tools that Bell Labs' software community quickly employed in the software production and documentation process. The collection of tools is known as the Programmer's Work Bench [20]. This environment typifies the tool-box where numerous tools are readily available, even redundant, and are not directly relatable to one language, each other, or a development methodology. The application of the tools is up to the developers and users as they deem necessary.

The design philosophy for the Programmers' Work Bench (PWB) as expressed by Dolotta and Mashey provides a foundation applicable to any large software system development. The users should be using the system early in the process so their needs and problems drive the design. Each new feature should be as consistent with existing ones as possible in order to maintain an environment that is simple, coherent, and conducive to productive use [10].

The UNIX environment includes a small collection of graphics tools. The UNIX shell has various versions of three tools called GRAPH, PLOT, and SPLINE. GRAPH accepts abcissa and ordinate pairs as input and connects this list of points by straight lines. Some control is available by specifying arguments at the invocation of GRAPH. The output of GRAPH is a file of commands accepted by PLOT. PLOT performs the actual graphics production on one of several terminals whose control codes are known to the UNIX system. SPLINE is similar to GRAPH in that it is a filter, or intermediate processing step, that produces a file that can be input to PLOT. In this case, SPLINE connects the input points with a smooth curve [21]. The use of these tools is depicted in Figure 1.

UNIX incorporates the graphics capabilities in a manner consistent with the other tools in the environment. The graphics is oriented towards the traditional use of graphics for plotting numerical data and is not very extensive.

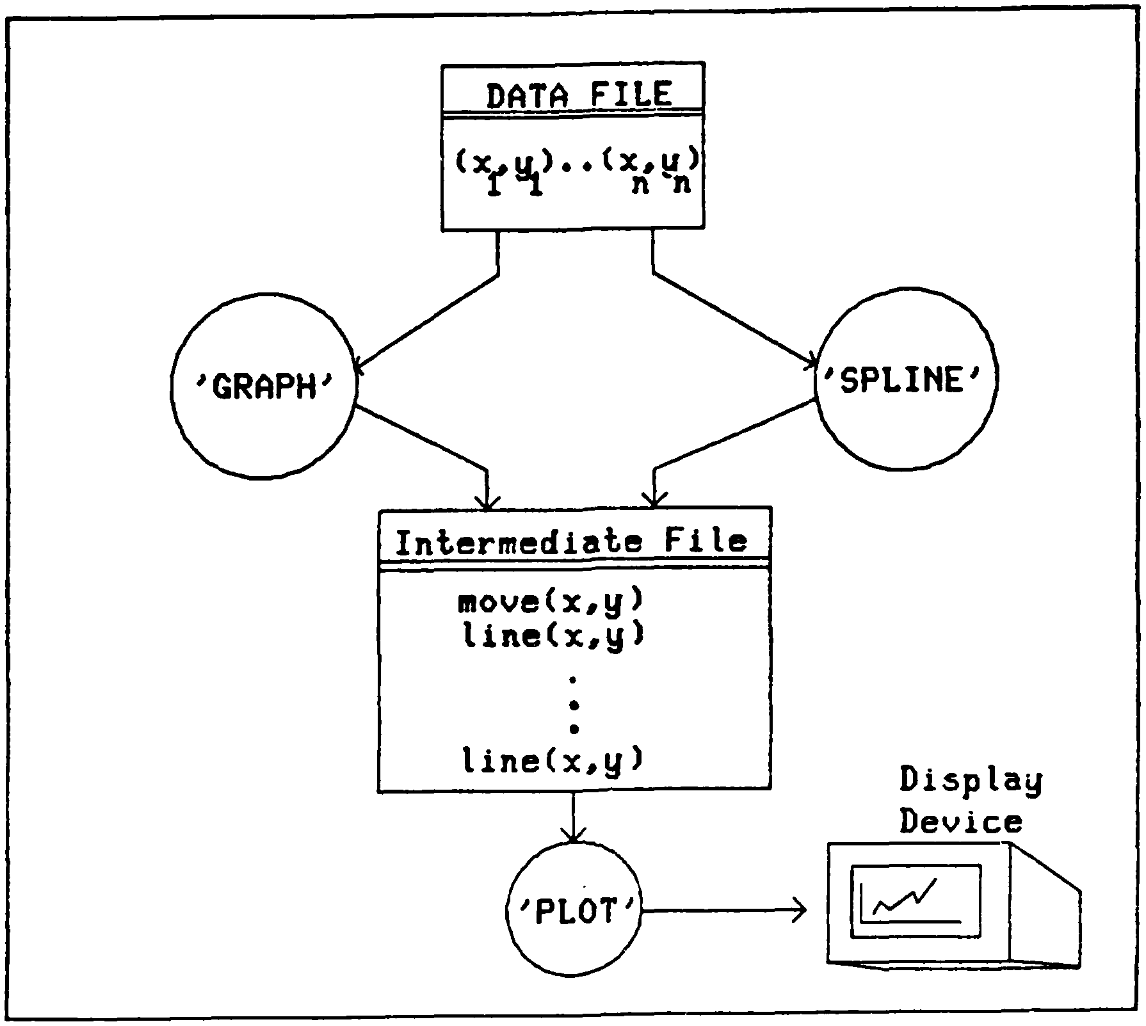


Figure 1. Graphics Tools in the UNIX Environment

The Ada Environment

The programming language Ada and the proposed Ada environment exemplify the second environment form. The environment was fashioned strictly to support Ada software development. The tools in the environment are integrated for the specific purpose of supporting one language. The tools are defined for several levels of implementation, are not redundant, ana are useful only for the development of software in Ada.

The following summary of the Ada environment is from a review of the Ada language and environment by Wegner [11]. Figure 2 depicts the organization of the Ada environment. The environment is structured into four levels, where each level is a layer that encompasses the capabilities of the layers below it, as well as adding new capabilities. Level Ø is the bare machine with its operating system and is the minimum required to build the Ada environment. Level 1 is called the Kernel Ada Program Support Environment or KAPSE. This is the machine-dependent interface to level Ø. The KAPSE is the interface that provides portability of the entire Ada environment by hiding the host-specific functions at this level.

The Minimal Ada Program Support Environment or MAPSE provides the recognizable tools for coding and execution of Ada programs. The minimal toolset inlcudes compiler, debugger, linker-loader, and a job control language interpreter. At the topmost level, level 3, are the Ada Program Support Environment (APSE) extensions to the previous level. Whereas the MAPSE supports all users, the tools of the APSE are specific to the users' applications such as simulators or testbeds or graphics.

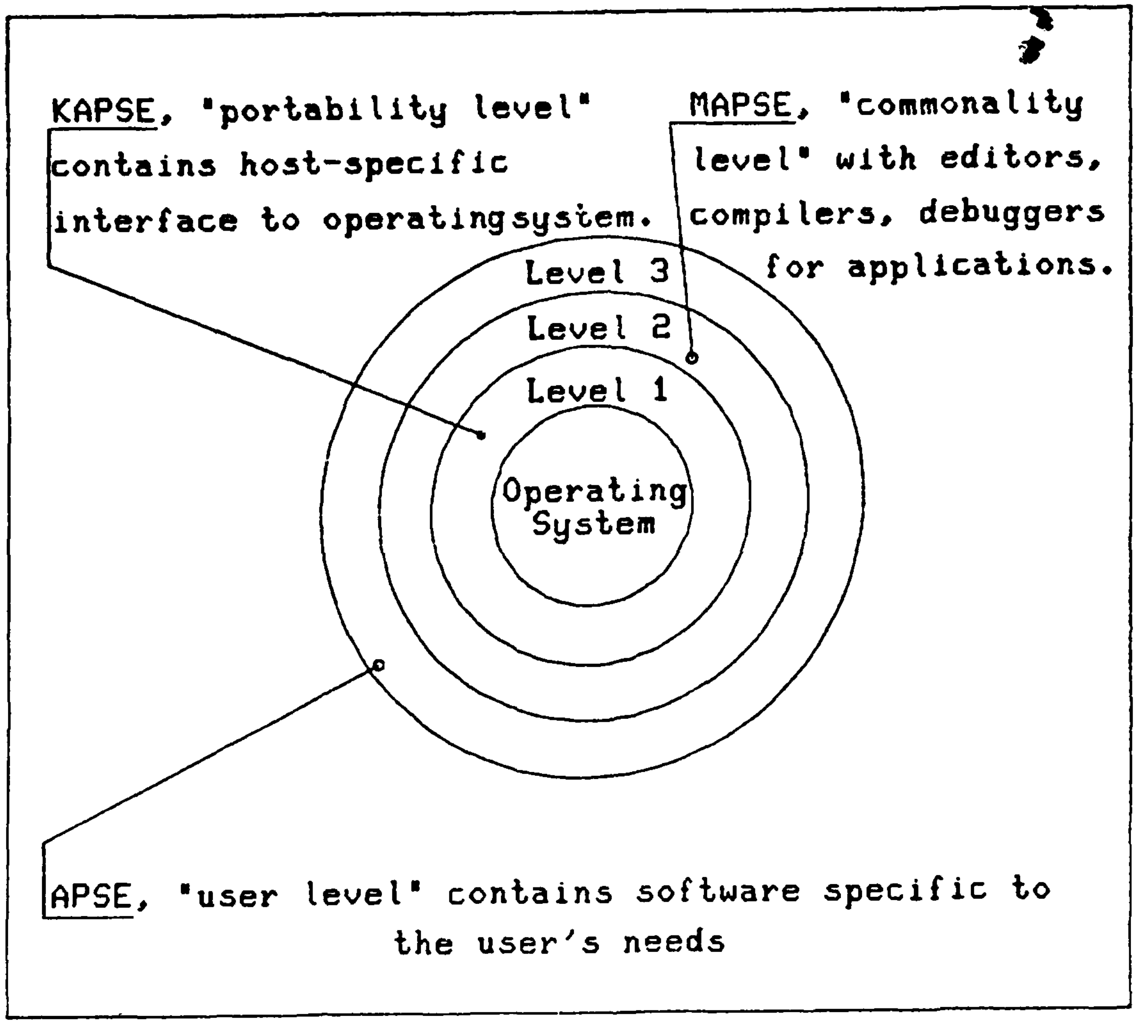

KAPSE, "portability level" contains host-specific interface to operatingsystem.
MAPSE, "commonality level" with editors, compilers, debuggers for applications.
Level 3
Level 2
Level 1
Operating System
APSE, "user level" contains software specific to the user's needs

Figure 2. Levels of Implementation of the Ada Environment

The layering of the Ada design has an important implication for all environments that require portability and levels of implementation. The environment is structured into a "portability level", a "commonality level", and a "user level." This allows convenient levels of implementation designed to allow the user to build onto the outer-most level without being concerned wih the lower level of implementation details.

The Smalltalk Environment

The Smalltalk environment is the latest version of a research environment which has been in development since the early '70s at the Xerox Corporation's Palo Alto Research Center. This environment differs from the UNIX and Ada environments in several respects, most notably its use of graphics. The Smalltalk system is based on a highly advanced version of a personal (nonshared) computer with local disc-based memory, virtual memory management, and a high resolution graphics display with up to 80K words of memory for the display only [1:178]. The graphics is not an added-on feature, the graphics kernel provides the principle user-machine interface.

The Smalltalk environment and language treat all interaction on an object level. Any object accessible to the user is capable of being presented for observation and manipulation without the loss of information of the state of other objects [12]. That is, when the compiler or editor or file manager is requested, it is treated as an object of current interaction with the user. The means of presenting an object for interactive manipulation is where the graphics plays a significant role. All objects are presented in windows defined on the display surface. If another object is desired it can be added to the display surface in another window. The windows on the view screen are analogous to sheets of paper on a desk. To move from one object to another, one window is selected to be placed on top of all the other windows. The contents of the windows remain unchanged when overlapped by another window. This is very useful because it eliminates the "modes" familiar to all programmers. For example, consider the problem that occurs during a compilation of a program. If an error occurs, the programmer needs to review the source code to see the cause of the problem and fix it. This may require terminating compilation, and entering the editor and locating the source code in question. In the process of doing this, the error message and state of the compilation is lost and cannot be recovered while in the editor mode. With Smalltalk, the compiler's status and messages appear in a window. When the editor is requested it appears in another window; to switch between compiler and editor only requires using a pointing device to pick the window to be displayed on top of the others and no information is lost. The corrected source code can be compiled by simply selecting the compiler's window and requesting that compilation be resumed or restarted.

The lessons from the Smalltalk environment indicate that graphics can be successfully integrated into the environment and serve as the primary medium for user-machine interaction [13]. With the use of graphics in the Smalltalk environment, interaction is accomplished a screen-full at a time.

Tool Design Requirements

As is clear from the preceding sections, the design of useful tools and environments can provide guidance for development of a large, interactive software system. Many of the concepts used in environment development are derived from studies of interactive conversations between user and machine. The important addition of concepts such as portability and the use of modular and easily combined tools result from the desire to produce software for a context much larger than an interactive program. The guidelines for tool development will be explored further for eventual application to the graphics environment.

The most significant point made by reviewers and designers of environments, graphics or otherwise, is the need for portability. The option for re-hosting a software tool with a minimum amount of effort markedly improves the utility of the tool. Portability of the tool assumes a second meaning as well. The applications developer becomes "portable" in that he can easily transition from installation to installation and have the same capabilities, at least in software, at each installation.

The portability of the software is the result of the design and implementation of the code so that it is machine-independent as much as possible. This requires minimizing the interplay between the tool code and the operating system support, typically file management and input/output facilities. As a result, these requirements demand some form of device dependent code. The device dependent software should be isolated from the device independent software and all information should be transferred across a well defined interface. The user should not be concerned with the underlying aspects of the tool they need to use. The developers and users are concerned with functionality at the application level [26].

Among the primary concerns for software developers is the need to provide a product that is taken seriously by the users, and while this may seem obvious it is really a critical factor [27]. An important criteria for development of new tools is its usefulness to users. Users are willing to learn to use a tool if they feel it is important to their own professional or personal ambitions. A significant aspect of usefulness is the ability to easily learn to use the tools. The time necessary to learn impinges on the developer's time to get the development job done, therefore the learning time must be minimized. The environment must be user-centered, helpful and supportive [14]. The most common means of help and support to a user at a terminal comes from the documentation and on-line assistance.

The documentation must have sufficient breadth and depth to cover all aspects of the tool applications. Reference material that is incorrect due to aging is a common source of frustration for the user. When the software is modified during the course of its use, the documentation must also be modified to reflect the current functions and operations. This same requirement applies to the on-line help libraries as well.

The on-line assistance should distinguish among different classes of users. Wasserman distinguishes three groups : the novice user, the casual user, and the expert user [26]. The on-line assistance should allow each class of user to obtain the information he needs, no more and no less. For example, the novice user may need a command description, the format for the command, its parameters, and an example to refresh his memory, while the expert user may need only the format. This requirement may be more useful when the software is an applications program used by non-programmers. Nevertheless, a help facility will be required for the tools in a research system if only to demonstrate the methods of developing an on-line help mechanism. Another useful mechanism for a help file would include a means to accumulate statistics on the frequency of requests to the help files. This information provides feedback to the developer to improve the efficiency or clarity of aspects of the application.

Minimizing the workload for the user is consistent with Kernighan and Plauger's maxim that a tool use the machine. In the face of an error the software must not terminate abnormally so the user has no comprehension of what caused the failure or error. There are two means of minimizing this possibility. One method is to notify the user if any request on the user's part will have major consequences. File deletion, interactive program termination, or overwriting a file or display can be considered major consequences. The options to abort a command or confirm a selction help prevent frustration on the part of the user. A second method for handling inappropriate or erroneous actions is to trap the errors and display an intelligible error message. The system should be prepared for incorrect responses on every input.

There are three important points for consideration when developing the error message, it must be meaningful, prompt and to the point. A delay in producing the error message is likely to result in missing the message when it occurs or missing it entirely. Therefore, it should be possible to frequently test for errors. A long error message becomes tedious if seen several times and may also scroll the erroneous command out of view or off a CRT entirely. Rapid learning is possible with prompt, succint feedback in response to an error.

An important objective in the design of a collection of tools is that the tools be extensible and lack undesirable side-effects when interacting with the input or output files, applications, or tools. These requirements imply the use of sound software engineering practice in the design and implementation of the code. The levels of implementation encountered in the discussion of the Ada environment is a good (but not complete) prototype for developing a flexible graphics system. Additional capabilities are added on top of lower level functions to provide a system oriented for a particular use. The lower level functions are not redundant and interact with other low level functions and new capabilities in a well defined way.

The requirements for the graphics tools come from studies of interactive dialog between man and machine, from studies of software development environments and from software engineering design and implementation practices. These criteria will be applied to the graphics system to produce a graphics tool that can be used for a variety of applications and extended for further research in graphics. Table 1 summarizes the requirements for the graphics environment as derived from software tool design criteria.

REQUIREMENT	IMPLEMENTATION REQUIREMENT
PORTABILITY	ISOLATE DEVICE AND SYSTEM SPECIFIC MODULES, AND PROVIDE A SINGLE INTERFACE.
MODULARITY	INDIVIDUAL FUNCTIONS IMPLEMENTED AS COHESIVE PROCEDURES, MINIMIZE SIDE EFFECTS FROM INPUT TO OUTPUT.
EXTENSIBILITY	EXISTING MODULES COMBINED TO FORM NEW APPLICATIONS, NEW CAPABILITIES ADDED AS MODULES ON NEW "LAYER."
FACILITATE LEARNING	ACCURATE DOCUMENTATION, ON-LINE HELP FOR USEFUL APPLICATIONS.
CONSISTENT INTERACTION	FEW AND CONSISTENT INPUT METHODS, PROVIDE FEEDBACK AND PROMPTING.
ERROR HANDLING	CONFIRM COMMANDS FOR SIGNIFICANT CHANGES, ALLOW ABORTING A COMMAND, PROVIDE TERSE ERROR MESSAGES PROMPTLY.

Table 1. Tool Design Requirements for Application to the Graphics Environment

Graphics System Requirements

The components and capabilities for a general purpose graphics system must be established. The diagram in Figure 3 presents the key elements of an interactive graphics system. The system is composed of hardware for interactive input, computation, display, and hard-copy. The software provides control of the graphics system, establishes the interface to the applications program, handles the input and output device allocation, and provides some or all of the graphics computation depending on the capabilities of the display hardware.

For the purposes of this investigation, the hardware components of the graphics system are already in place and influence the design of the software. However, the software should support the equipment already available and be sufficiently flexible to accomodate new equipment. To insure this generality, the graphics system capabilities in terms of the general hardware and software requirements will be presented in the following sections.

Hardware Environment

The components required for the hardware portion of a graphics system depend on the intended application. For a general purpose graphics research system the hardware components should have a large range of capabilities to demonstrate the trade-offs of one implementation approach or another. The system should also support new, prototype devices as part of the research effort. There are four functional groups of hardware required for interactive graphics work: the graphics displays, input devices, output (hardcopy) devices, and

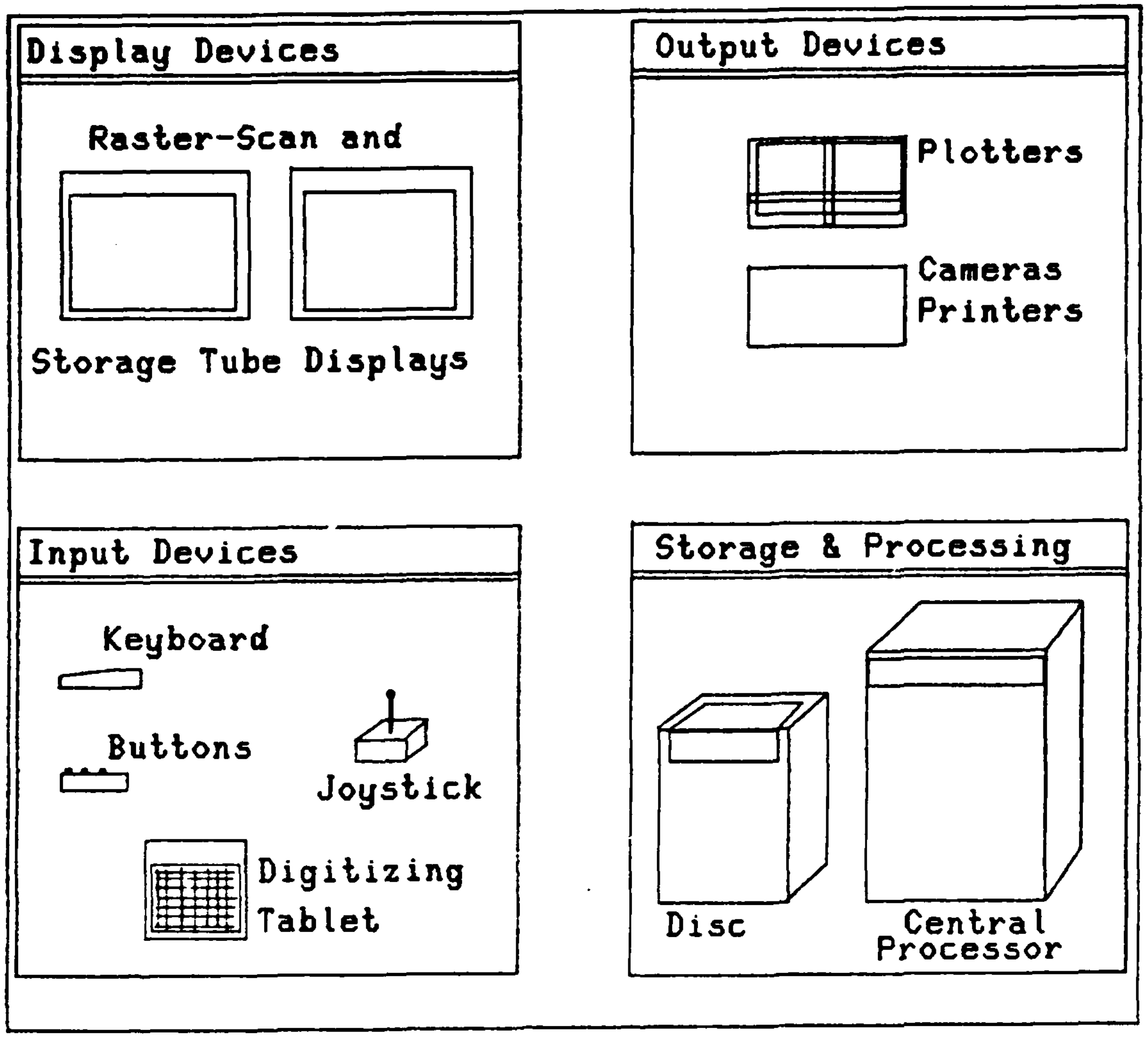


Figure 3. Elements of an Interactive Graphics System

processing and storage devices.

The first component to be considered is the display. A general purpose facility should support devices with a range of resolution and color capabilities. The resolution should be 40-200 points per inch [22] which, for an "average" display size of 19 inches diagonally, translates to 512 X 512 to 4096 X 4096 addressable points. Low resolution devices with 256 X 256 points or less are available as well as the extremely high resolution devices with 8192 X 8192 display points. The displays should have representative capabilities for monochromatic, gray-scale, and color presentations. Typical gray-scale devices allow 64 shades of gray and the color devices allow 8 to 64 simultaneous colors from a pallette of 8 to 4096 possibilities [23].

A set of input and output devices are required for establishing an interactive graphics capability. The interactive input devices were first categorized by Foley and Wallace as the keyboard, pick, button, locator, and valuator devices [15]. These devices are sufficient to accomodate the input requirements of most applications. The pick device points at a user-defined object like a line or polygon. The light-pen is the prototype pick device. The button input device is for selecting a system defined object such as a foreground color. The locator is used to indicate position of an object in the user's drawing space. The graphics tablet is a typical locator device. The valuator is used to determine a single real number value. A potentiometer set by means of a thumbwheel is a typical device used as a valuator.

The output devices required for a graphics system depend, again, on the application. Since the objective here is to provide a means to utilize the available graphics devices in a general purpose system, several possible output devices will be discussed. The output devices, besides the display, include a large variety of pen plotters, dry-paper copiers, and dot-matrix printers with a graphics capability. The newer output devices capitalize on the display devices' properties for intensity control and shading by photographing the display surface on a variety of film sizes. The display may also be fed to film or video recorders that can be edited to produce animated movies produced with computer graphics. Dot-matrix and ink-jet printers can also be used to produce color hard copy. A phototypsettter can be used to produce publication quality text and graphics.

The processing power and secondary storage capabilities are important areas of consideration for the graphics system. While the host computer provides the system services like an editor, file manager and secondary storage management, the graphics peripherals will have varying amounts of processing hardware, memory for the display generator, and additional processing units for control and instruction execution. The simplest device may have memory sufficient only to provide the display generator with information for the refresh cycle, whereas the typical "smart" display devices are actually stand alone computers dedicated to graphics. These devices have hardware for clipping, image transformation and vector generation, program memory for user definable subroutines, and storage for display files on local secondary storage.

System Interaction

The final requirements are not specifically related to hardware but to the system in general. First, the interaction must not al.ow what Foley and Wallace called panic and frustration to occur [15]. Panic occurs when the system fails to respond in a timely manner to the command. This requires that the interaction rate be considered in system design. The output rate should not force the user to spend time wondering if the system is hung-up in an endless loop, or if an action was taken that was not observed or desired. The system should respond in 0.5 to 1 seconds for simple commands, within 3 seconds for more complex operations and the production of "typical" complex displays should not exceed 15 seconds [29]. The system must accept input rates that have great variability: a tablet can generate a 1000 abcissa and ordinate pairs in a few minutes, while one new frame request may be generated in a half hour when the image is the subject of group discussion.

On the other hand, frustration occurs when the user cannot easily accomplish the desired operation, or receives an unexpected response, or cannot easily recover from an error. These considerations require that the devices constitute a system that is compatible at every device interface. For example, a display may contain 100 pickable points in a small area on the display surface. The light-pen or other pick device must have a compatible picking resolution.

Software Requirc nts

The software for the general purpose graphics system must allow for the simplest graphics hardware as well as the most advanced devices in order to accomodate the possible combinations of graphics peripherals. The simplest peripherals require more software functions than the graphics equipment with the built-in graphics hardware. For completeness, the requirements for the software will not presume any advanced graphics peripherals are available. When more capable displays become part of the environment, the required software becomes a subset of the previous software.

The fundamental drawing capabilities allow positioning textual information and plotting points and lines. Control over cursor movement must also be provided to allow selecting a starting point without the cursor movement from the previous position to the current position being recorded in the picture. These capabilities are the required graphics primitives and are necessary for any graphics drawing [1:33]. However these capabilities are generally not sufficient for most applications. These primitives are combined within a data structure (called a segment) that allows subsequent manipulations to be performed on a collection of primitives. Combinations of the segments establish an applications model [1:30] that allows the user to produce graphics images in terms of the logical segments rather than a collection of MOVES and LINES.

The software must also provide control over the graphics devices. For a general purpose graphics system the software handles the input devices, controls the output device selection, controls the display selection and picture modifications. The software also provides the modelling and transformation capabilities as part of the processing requirements. The software must be capable of accessing secondary storage either explicitly or through the host file management system.

These three components, the picture description at the graphics level, the picture description at the user level, and the interaction methods, constitute the "programmer's conceptual model" of an interactive graphics system [1:29]. The software is required to transform the application data structure into a picture according to the user's desires.

The implementation form of the graphics software is a major consideration for the graphics environment. The graphics software for the general purpose system must support development of both additional graphics capabilities and applications development. The latter requirement implies that the graphics functional capabilities must be available to the applications programmer which further implies that the graphics software be accessible from the applications language.

The capabilities required for the hardware and software of the graphics system are summarized in Table 2.

HARDWARE		SOFTWARE	
DISPLAYS	COLOR, RASTER-SCAN DISPLAY, STORAGE TUBE	PRIMITIVES	LINES, POINTS, CURSOR MOVES, TEXT
INPUT DEVICES	BUTTONS LOCATOR VALUATOR KEYBOARD PICK	SEGMENTATION	LOGICAL GROUPING AND ASSIGNMENT OF NAMES TO PRIMITIVES
OUTPUT DEVICES	PLOTTERS CAMERA PRINTER	CONTROL	INPUT HANDLER, OUTPUT DEVICE SELECTION, AND PICTURE CHANGE CONTROL
STORAGE & COMPUTATION	SECONDARY MEMORY AND CPU		

Table 2. Graphics System Requirements for Application to the Graphics Environment

Summary

The development of programming environments arose from the need to improve the productivity of the software developer. The strategy for developing a graphics workbench must therefore be consistent with the strategy behind programming environments. The development of a general graphics capability provides a graphics work bench for development of graphics tools as well as the pictures themselves.

The graphics environment consists of a display devices, means for input and hard-copy output, and a variety of processing and storage capabilities. The software provides the control for the graphics devices and manipulation of the primitives and their data structures. The software must be devised to support the hardware, the graphics primitives and be sufficiently general to support both additional graphics capabilities and applications programs.

In the next chapter, the general strategies of environment and tool design are expanded into specifications for the qualities of the graphics environment. The hardware to be used in this investigation is discussed. The alternative forms for the implementation of the software are presented along with the attendant advantages and disadvantages of their designs.

III. Design Specification for the Graphics System

At the outset of this investigation, it was intended to satisfy many users and applications. Hence the design is for a general graphics capability that is capable of being extended to meet the user's needs. The hardware components of the AFIT and Avionics Laboratory were already in place at the outset of this investigation. These hardware constraints are presented in this chapter and the impact on the design is assessed. The options for software development and extension are more numerous. The various software implementation structures are discussed. In the second section, the environment and tool development requirements of the preceding chapter are applied to the graphics system design and the areas for further development are identified. The chapter concludes with testing criteria for the design and a proposal for a program to be used as a tool in the graphics environment.

The Graphics Hardware Constraints

The components of the graphics hardware systems at AFIT and the Avionics Laboratory's Electronic Warfare Analysis branch are presented and discussed in light of the requirements of the preceding chapter.

The host computer system provides the nucleus for developing a graphics system. One of the most popular hosts is the Digital Equipment Corporation VAX 11/780 which is classed as a "super-mini" because of the combination of its small size and extensive capabilities. The VAX architecture uses a 32-bit word size and a virtual memory management system. Up to 8 megabytes of physical memory is installable, and several varieties of high capacity secondary storage systems are available. The Avionics Laboratory and AFIT have VAX 11/780 computers which supply the required processing and storage capabilities for the graphics system.

A variety of peripherals are attached to these host computers. The display devices common to both systems are from the Tektronix 4010 series of direct view storage tube displays. These include the 4010, 4014, and 4016 terminals which have display dimensions of 10, 14 and 16 inches measured diagonally. These devices satisfy the requirements for high-resolution (780 X 1024 addressable points) monochromatic displays.

In addition to the 4010 and 4014 displays, AFIT has an Intelligent Systems Corporation color raster-scan display (ISC 8001). This device has 8 fixed foreground and background color capabilities. This is a low resolution device: the CRT measures 19 inches diagonally and has 159 (vertical) by 192 (horizontal) addressable points or approximately 20 points per inch. However, in some instances, the display is limited to 80 by 48 addressable "blocks", which is equivalent to the text resolution capability of the display. The Avionics Lab has a RAMTEK 6210 high resolution (512 X 512) color raster-scan display. This device can display 16 colors at one time from a pallette of 64 possible colors. Neither system has displays that provide gray-scale renditions nor are there any devices that supply the very high resolution. AFIT is planning to procure an Evans & Sutherland PS300 graphics system which will have a very high resolution color display on the order of 4096 by 4096 addressable points [41]. As stated in the requirements, the graphics software should be designed to capitalize on a variety of display capabilites.

Somewhat less available are the required interactive input devices. The RAMTEK system is the most complete in that it has hardware to support nearly all of the input devices specified in the requirements. The RAMTEK has a Summagraphics digitizing tablet, a RAMTEK supplied joystick and the keyboard includes 16 user-definable function buttons. All of the Tektronix terminals include the thumbwheel cursor controls with the keyboard as standard equipment. The ISC has button selection capability on the keyboard for foreground and background colors.

The other output devices are varied as well. The Tektronix displays have a capability for hardcopy paper output by copying the screen to a dry paper copy device manufactured by Tektronix. The Ramtek system uses a camera to photograph the CRT display on color 35 millimeter film. The AFIT system has a Printronix dot-matrix printer with a built in graphics capability, and both facilities have Calcomp drum plotters.

The hardware meets the requirements for a graphics environment as previously presented in Figure 3 and Table 2 of Chapter 2 and covers the capabilities spectrum, although there are deficiencies that will require further consideration. For example none of the peripherals includes a pick device. This situation is readily overcome by simulating one or more of the interactive input devices or output capabilities through software[1:197-214].

The Graphics Software Design Alternatives

Both AFIT and the Avionics Lab have applications and graphics software that suit each particular system's requirements and produce the desired results. This graphics software will be considered for the proposed graphics environment in addition to other software that has been obtained but not used. The existing graphics software takes two forms: a graphics language and a library of subroutines. There are advantages and disadvantages to either format depending on the application. These two forms for the software will be discussed in light of the requirements and the hardware available to determine which form is to be used.

Graphics Languages

The most common use for a graphics language is to drive a particular manufacturer's display. The languages have their own syntax, facilities for address specification to local memory, and commands for both graphics and control of the terminal and its interfaces [29]. Not all graphics languages are vendor supplied; general purpose graphics languages have been proposed and implemented. These languages may be extensions to an existing high level language or a new language requiring its own compiler or interpreter.

Graphics Subroutine Libraries

Graphics software is also available in the subroutine library format. These software packages are typically implemented as a collection of subroutines that are compiled and used as an object library. In this way, the applications program is written using the graphic's package function calls which are linked to the applications object code to form an executable object. Many graphics packages are available commercially that use the library of subroutines approach to provide the applications programmer access to the graphics capabilities for particular devices.

The growth of graphics applications and devices to support high quality graphics has led to a proliferation of graphics software. The features of many of these graphics systems have been incorporated into proposed standards. The trend towards standardization of graphics capabilities uses the collection of subroutines approach. At present, there are two proposals for a standard graphics package capability. The Core standard, proposed by the ACM SIGGRAPH, is under review by the American National Standards Institute[3]. The other contender for standard status is the (principally) European proposal called the Graphics Kernel System (GKS) which appears about to be adopted by the International Standards Organization [18, 33]. The same GKS proposal is under review at ANSI as the standard for the Programmer's Minimal Interface to Graphics (PMIG) which primarily covers workstation control and ouput for two dimensional graphics [25]. The goal is to produce a set of features that "can be reasonably easily supported by most existing graphics hardware and cannot be easily built on top of other (standard) capabilities" [3:II-15]. The standards provide input and output primitives, segmentation, transformation, and control.

The Available Software Options

The RAMTEK terminal uses a graphics language for its operation [29]. The corporate proprietary Color Graphics Language has over 90 commands for both graphics production and control of the display and the its input/output devices. The commands have their own syntax and are interpreted by firmware within the terminal. The CGL is the only graphics language readily available for use in the Avionics Lab graphics environment. The remaining software options are all subroutine libraries.

One of the most common display device and graphics software combinations is the Tektronix terminal and the Plot1Ø graphics library. The sheer number of installations have made the Tektronix 4Ø1Ø series terminals, until recent advances in raster display technology, a de-facto standard. It is interesting to note that the newer, raster scan color displays typically include a Tektronix emulation mode at the expense of the color and refresh capabilities. Plot1Ø is suitable for any two-dimensional plotting and provides some input capabilities when used in conjunction with the thumbwheel devices on the keyboard.

Another package available was MOVIE.BYU which is produced at Brigham Young University and the University of Utah. MOVIE.BYU is not implemented as a library of subroutines, but as a collection of individual programs that are executed as needed to produce the desired output. While it is purported to be a general purpose graphics capability, it was designed specifically for use with finite element analysis via graphics simulation. MOVIE incorporates advanced features of graphics software like generation of continuous tone images and hidden line and hidden surface removal [28]. MOVIE also incorporates interactive input for image development.

The desire for a general graphics capability for use with the variety of host and peripheral combinations led to the consideration of the implementations developed to the specifications in the ACM Core standard. Two particular packages were available for implementation on the VAX 11/780 at the Avionics Laboratory and AFIT. The software from the Lawrence Livermore National Laboratory (LLNL) in California was obtained by Tarbell and is an implementation based loosely on the ACM Core standard [30]. It was originally developed on the large mainframes at LLNL and partially recoded for the VAX. The files that were obtained, unfortunately, were not complete. Additionally, not all of the source code had been completely changed over to VAX FORTRAN [8]. The LLNL implementation consists of two functional components, GRAFCORE and GRAFLIB. GRAFCORE is the Core standard set of functions as implemented at the laboratory, while GRAFLIB is an extensive set of additonal capabilities that provide a number of useful tools for the scientific community at Livermore. These library functions include plotting in a variety of coordinate systems, arc and contour generation, interpolation and so on. The structure of GRAFCORE/GRAFLIB and its application to this investigation will be discussed later in this chapter.

The second implementation of the Core available on a VAX was developed by the George Washington University, Washington, D.C. This implementation includes FORTRAN source code for implementing the Core specifications of output primitives, picture segmentation and naming, attributes, viewing operations, and control. No input routines had been made available. GWCORE is a strict implementation of the Core with little embellishment. The GWCORE source code was also supplemented with very good documentation in the form of a User Document [39] and an extensive Maintenance Document. In addition, the test cases supplied with the software worked with the Tektronix displays.

Both GWCORE and GRAFCORE are still in development at their respective laboratories. A new release of GWCORE has been supplied to the National Energy Software Clearing House by George Washington University complete with the input routines as specified by the Core software proposal. The newest version has been ordered for AFIT from the clearing house. GRAFCORE is still forthcoming.

Application of the Design Criteria to the System Specification

At this point, the components of the hardware for the graphics system and the alternative forms for the graphics system software have been presented. It would be possible to implement the desired capabilities by extending the existing software. The available software has also been briefly discussed. This section uses the criteria as developed in Chapter 2 as the strategy for shaping the graphics environment into a tool for applications development. The selection of software will dictate the utility of the graphics system. The initial step in the development will be to define the advantages and disadvantages of the graphics language or subroutine library form for the graphics software.

One advantage of the graphics language is that it is able to more precisely convey the desired graphics function than the collection of subroutine calls. The graphics language is more easily read by the author and user than a sequence of subroutine calls with a clumsy parameter list construction. The graphics language for a terminal, like CGL for the RAMTEK, allows the language to take full advantage of the device's capabilities. However, the diverse graphics capabilities of the systems described above point out the need for designed-in software portability through device-independence. Lack of portability is the principle disadvantage of a graphics language. The graphics language approachs have seldom progressed beyond machine-specific implementations devoted more to research in computer languages than to graphics. The languages that are extensions to an existing programming language involve changing the compiler which can be an expensive and complex process. Furthermore, in a production environment, the overhead that results from the more extensive compiler options effects all users regardless of whether they use the graphics features [34].

Learning a programming language that has been extended or a language dedicated to graphics thwarts the need to limit the learning time for a new utility [19]. Admittedly, even with a graphics package composed of subroutines some procedure calls will need to be learned, but it is considerably easier to learn a procedure calling method than to learn an entire language. Subroutine packages do not require changes to the compiler or language, instead they effect the runtime environment. The compilation, linking and execution processes demand considerable time and space from the host computer. On the other hand, the use of a library of subroutines allows the applications developer to use the high level language of his choice as long as it allows reference to procedures and functions written in another language. All efforts at producing a standardized graphics capability have used the library of subroutines concept as the principle implementation method [25]. One of the stumbling blocks of using a standard package is its inability to avail itself of the more advanced devices with built-in graphics functions. However, in the general purpose system under consideration for this investigation, a standardized capability satisfies many of the requirements discussed so far.

The objectives for development of a standard graphics capability encompass many of the objectives for development of the graphics tool. The ACM Core standard proposal is discussed to highlight the compatibility and deficiencies with the specification for the desired graphics system. The numerous contributors to the Core porposal sought to produce a "functional, clean and consistent" standard graphics capability and this is in keeping with the desires for producing a quality software tool.

The areas to be presented are the main categories of the Core proposal itself. An excellent descriptive summary and glossary of graphics terminology appears in the ACM Computing Surveys special issue on graphics in December 1978. The article by Michener and van Dam provides an overview of the functional capabilities of the Core [35].

Output Primitives and Attributes The objects as presented on a display or in a hard copy are a collection of output primitives that have specifiable attributes. Lines, text, and polygons are a subset of the primitives available in the Core. These primitives possess attributes such as linestyle, linewidth, color, character size, character spacing, polygon fill color, edge style and more. The attributes are initialized to a default value or set by the applications programmer. Thereafter, every instance, or ocurrence, of the primitive has its attributes set to the current value.

The attributes include the capability to scale a primitive (or segment, to be discussed below) up or down in size, rotate it about its origin, and translate it relative to its origin. These image transformations are distinct from the viewing transformations.

Viewing Transformations The view of the object is specified by the current values of the viewing transformation equations. The viewing transformation maps the world model to the normalized view surface coordinate system, where the view surface is a rectangular two dimensional area. The second transformation maps the normalized coordinates onto the display surface within the currently specified (normalized) viewport. The use of a virtual display surface, as the normalized rectangular area is called, allows the normalized-coordinate-system to device-coordinate-system mapping to account for the variability of the final, physical view surface resolution and dimensions. The normalized points are simply scaled by the device driver software.

The primitives may lie partially outside the currently specified world window and therefore require clipping before being passed to the device driver. That is, those portions of the primitive that lie outside the window are eliminated and not mapped to the viewport. Only the objects visible in the window are mapped to the screen.

Segmentation and Naming The picture displayed on the view surface is defined by one or more segments. The application program uses segments to change parts of the picture. A segment typically consists of several primitives, and like the primitives, a segment has attributes that apply to the entire segment as an entity. The segment may be named and all of the attributes altered by specifying the name. The segment attributes support visibility, highlighting, detectability by an input (pick) device and image transformations.

Input Primitives Interactive input devices are called logical input devices because, as was mentioned earlier, some devices may require simulation. For example, the joystick is considered a locator device but can easily be used as a pick device through the appropriate software routines. The means for interactive input include the keyboard, pick, valuator, button and locator devices.

Control The general operation of the Core system as a library of routines requires initialization of the Core parameters, selection and initialization of view surfaces and proper termination of the Core. The control section of the Core has responsibility for these functions as well as control of picture updating when a change is requested. The updating of the device is device specific, for a refresh display the change is immediate, on a direct view storage tube the screen is erased and the changed scene is redrawn in its entirety.

The Core system satisfies many of the specifications for a general graphics capability. The principle concern of the Core was portability of the graphics programs and programmers. The implementation of the core must isolate dependent interactions of the host operating system and graphics peripherals. The Core design provides the primitives, attributes, segmentation, viewing transformations, input and control functions discussed in the requirements.

The use of a graphics language that was designed for a particular device prevents it from being useful for the general purpose graphics system desired. The graphics languages that are extensions of other languages or developed specifically for graphics are not realistic options for development because they have not been widely accepted as applications languages, and in fact are rarely found outside a research group. Of the subroutine libraries available, Plot1Ø is not useful for displays outside of the Tektronix family and does not include functional capabilities to the extent the Core does. MOVIE.BYU has extensive capabilities but lacks the general applicability of the Core standard. Also, as was previously stated, MOVIE.BYU is not implemented as a linkable library and is not accessible from general applications programs. The capabilities of MOVIE.BYU beyond those of the Core make it an attractive source of eventual enhancements to the general purpose software system.

Furthermore, since two implementations of the Core are available, the Core was selected as the basis for the graphics environment. The GWCORE package was selected because it was functional on a VAX, came with a driver for the "ubiquitous" Tektronix, had sufficient documentation to allow further development and extension, and is a clean implementation of the standard proposal.

The primary deficiency of GWCORE was the lack of the input primitive functions. This deficiency is not a severe impediment to development since the input routines had been previously designed by Curling [7]. The design uses a structured methodology to define the data structures and process structure for a two-dimensional, synchronous input capability. The addition of this capability will be accomplished for this development effort.

The design of the Core system also satisfies the specification for consistent interaction with the user in the face of an error. Every functional capability has a description of potential errors that consists of a standardized error number, a severity code, and a one line statement of the problem. In an interactive application, this error reporting provides sufficient information to clear the error and resume interaction. The severity code provides a qualitative indication of the likelihood of continuing to produce the desired output with the desired result after the error has occurred. This facility gives the user control over termination and does not automatically abort the interactive session [3].

The combination of the Core standard output implementation by the George Washington University and an implementation of Curling's input routines will provide the central capabilities for the graphics environment. Several requirements remain to be fulfilled, namely, the on-line help aid and a user's manual, device drivers for additional devices (especially a raster device), and additional tools to improve the quality of the general graphics system and test its ability to be extended and used for applications. Additional software, such as that provided by Livermore Laboratory's GRAFLIB, can be added on top of the GWCORE package to provide a "user level" collection of tools that can be assimilated into applications programs.

Testing

With the selection of GWCORE as the basis for the graphics software, the testing will begin with the software developed to expand the Core capabilities to include a color display and the input routines. The primary means of integrated testing will be to develop an application program that exercises the capabilities of the Core implementation and its environment. The central specifications to be tested include the following:

1. Device independence between graphics peripherals in one environment.
2. Implementation of the input functions and integration with the core.
3. Error trapping ability of Core and applications programs.
4. Utility of help functions and documentation.
5. Suitability of implementing additional layers on top of Core.
6. Ability to include additional primitives and attributes in applications.

The measure of device independence within one graphics sytem can be tested by developing several, relatively small applications programs and then executing them for each of the device drivers. The programs must execute correctly and use the peripheral's capabilities to the maximum extent possible. For example, an application that produces a color bar chart on a color device should produce a similar chart on a similar device. Likewise the chart should be reproducible on a monochromatic display and a hardcopy device as well. As was previously noted, the display driver supplied with GWCORE functions with the Tektronix terminals. To test the display device independence, another driver will have to be developed and the same picture reproduced on either display. This additional driver will also demonstrate the ability of the graphics software implementation to accomodate various display capabilities so applications software can make use of the best display for the given application.

The input routines must be developed to not only the Core specification, but to match the data structures and algorithms of the GWCORE implementation. Initial testing will require developing the input routines from Curling's design and having the routines pass the desired operation code and parameters to a stub that simulates the device drivers available with GWCORE. This will provide a test of the Device Independent side of the input routines and the device driver interface. The interface between the graphics software that is independent of the display used and the software that operates a particular display, is called the Device Independent/Device Dependent (DI/DD) interface. The stub will be replaced by the device driver software. The device driver will be tested separately by simulating the calls from the Device Independent side of the interface. Finally, the input routines must be integrated into the GWCORE library and inlcuded in an interactive application program.

The current literature and the concept of tool design suggested a suitable application to test not only the input routines, but would require developing a suitable interactive interface with on-line help, error handling and the remaining requirements as originally presented in Table 1. A computer-based editor is a tool typically used to prepare textual material whether a document or software. Recently, an interactive graphics editor was described that manipulates diagrams like a text editor manipulates words [17]. The Core version of a general purpose interactive graphics editor would be a general purpose tool if it could be implemented with standard Core capabilities and made consistent with the tool development criteria. Therefore, the application to be developed will be an interactive graphics editor. It should allow an unskilled user to learn to place simple graphics objects on a display, control the attributes of the picture, save and restore the picture to allow editing, and handle errors in a manner that does not discourage continued use.

Summary

The hardware components of the graphics system meet many of the requirements for a graphics environment although enhancements would provide more flexibility and adaptability for a variety of applications. The numerous software options were also presented. The graphics languages were dismissed from consideration because the device specific nature of the languages would prevent the generality and portability desired. The development of standards for graphics systems made the standards attractive candidates for the graphics capability. The ACM Core standard proposal was highlighted and shown to satisfy the requirements for both hardware and software desirable in a graphics system. On the basis of adherence to the Core specification and functionality, an implementation of the standard by George Washington University (called GWCORE) was selected as the software system for the graphics capability.

The deficiencies of this software were identified and remedies proposed. GWCORE lacks the input routines specified by the Core proposal and system requirements. The routines will be developed using Curling's design [7], and integrated and tested with the GWCORE package. Device drivers will be written for the available graphics peripherals. An application program, the interactive graphics editor, was proposed to demonstrate the use of the graphics software. The design requirements from software environment and tool design point out the need for additional developments. An on-line help system supported by sound documentation will be added to the application to demonstrate the usefulness of HELP commands and the the existing system utilities for HELP implementations in applications programs. Graphics "primitives" like circles and rectangles will be developed from the Core set of primitives. These modular additions provide added utility to an interactive editor and demonstrate the extensibilty of the Core implementation.

In the next chapter, the design, implementation plans and testing methods for the software is presented. The device driver, input routines and application programs are discussed in some detail in light of the requirements for a graphics tool.

IV. Implementation of the Graphics Software and the Tools

The previous chapter presented the reasons for selecting the Core standard for the graphics system and identified areas for development and testing. This chapter details the implementation and testing of the additions to the Core system. The three main thrusts of the implementation involved developing an additional device driver for a color display, writing the input routines to allow interactive picture development, and development of an application program; the latter to demonstrate the ability of the graphics environment to produce a useful graphics tool. The organization of these capabilities is similar to the Ada environment in the use of a hierarchy to support portability, commonality, and user level implementations. This organization is depicted in Figure 4.

Device Driver Development

The output-only Core implementation as obtained from George Washington University included a functioning device driver for the Tektronix 4010 series of direct view storage tube displays. The AFIT laboratory has a color display manufactured by Intelligent Systems Corporation (ISC). This display is a raster-scan device with a small (8K) refresh buffer controlled by an 8008 microprocesser. This is a display with a relatively old technology but does have features not found in the Tektronix displays. The ISC has the capability to display 8 foreground and background colors, highlight objects by blinking and double the height of characters. Developing a device driver provided an introduction to the Core algorithms and data structures and was

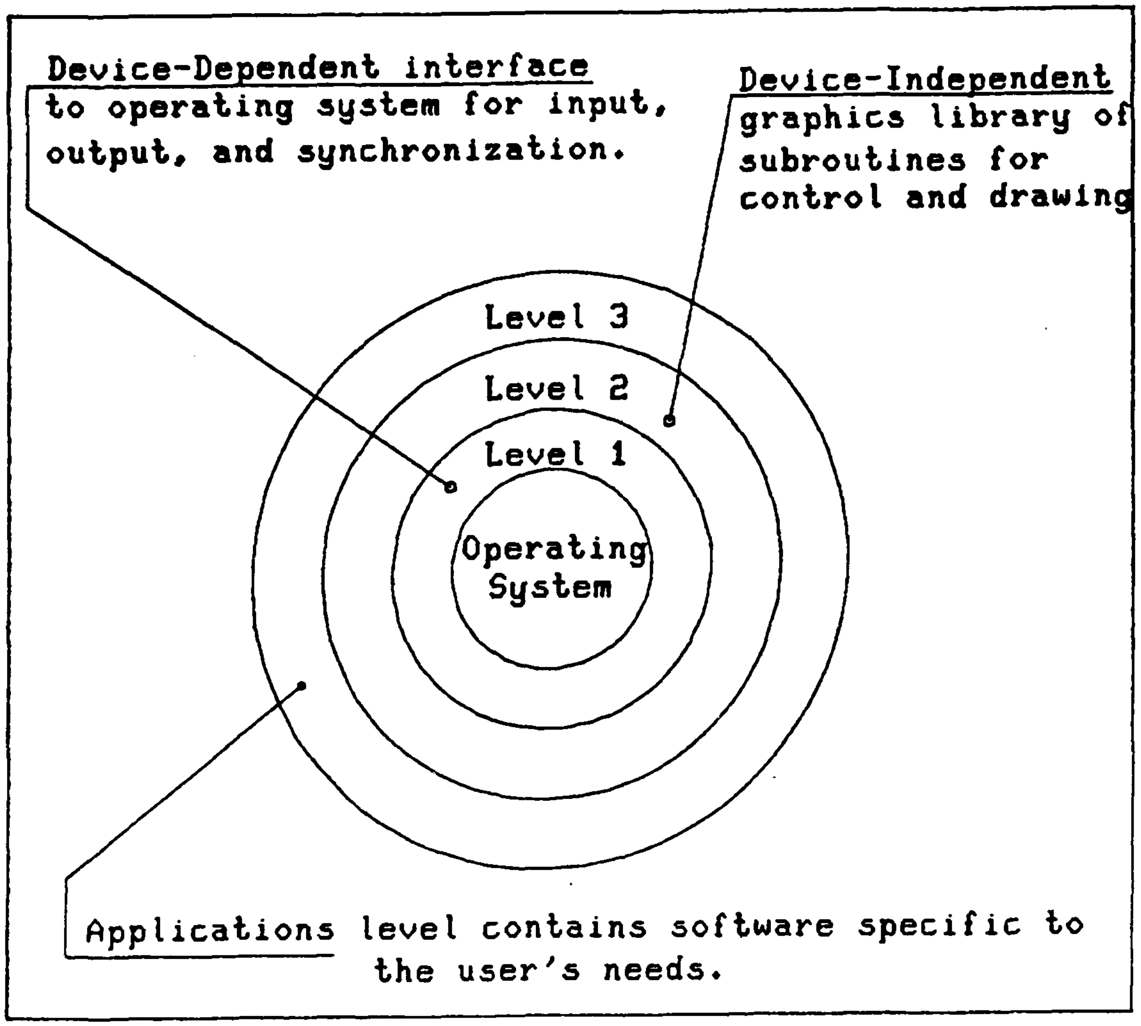

Device-Dependent interface
to operating system for input, output, and synchronization.
Device-Independent
graphics library of subroutines for control and drawing
Level 3
Level 2
Level 1
Operating System
Applications level contains software specific to the user's needs.

Figure 4. Levels of Implementation of the AFIT Graphics Environment

therefore undertaken as the first step towards developing a more complete graphics system.

Since the bulk of the graphics software is independent of any particular display's capabilities, a device driver provides the graphics peripheral with instructions that are peculiar to that display. Figure 5 depicts the GWCORE output organization of the device-independent front-end and device-dependent code generators. The device driver performs the transition from device-independent instructions (stored in the display file) to the device dependent instructions for the graphics device.

The ISC driver is patterned as closely as possible on the Tektronix driver. The Core standard software proposal does not include a specification for a standard device driver structure. It therefore seemed appropriate to model the new driver on the functioning Tektronix driver. This proved adequate up to a point. The similarities and differences of the two drivers are contained in Table 3. The driver for the ISC is somewhat more extensive due to the additional capabilities of that display for color, highlighting, multiple character fonts and so on.

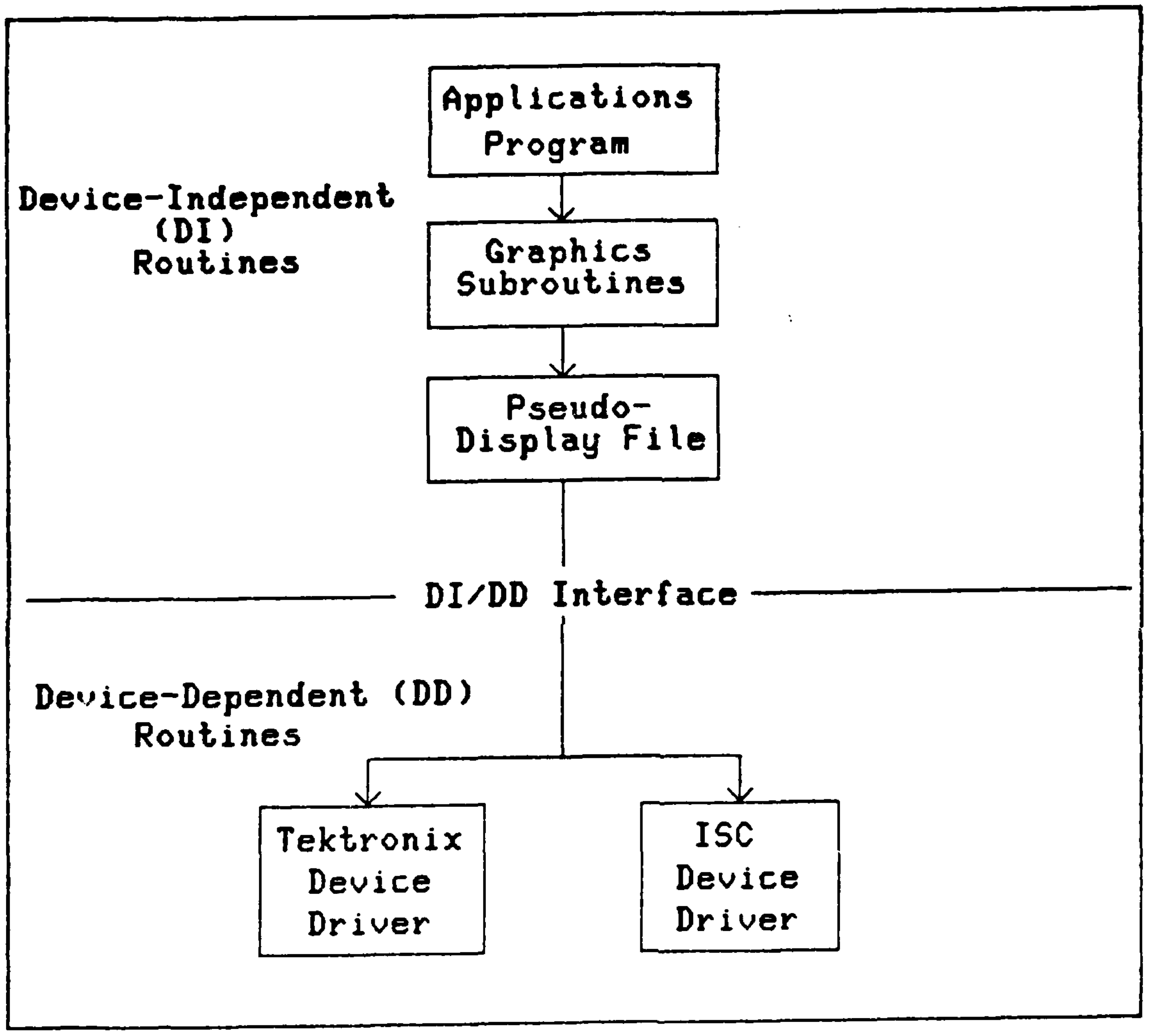


Figure 5. Organization of Device-Independent and Device-Dependent Code Generators

FUNCTION	TEKTRONIX	ISC
MOVE	YES	YES
VECTOR PLOT	YES	YES
TEXT	YES	YES
MARKER	YES	YES
COLOR	ON OR OFF	8 FOREGROUND/BACKGROUND
HIGHLIGHTING	NONE	YES
CHARACTER SIZE	ONE	NORMAL/DOUBLE HEIGHT
POLYGON FILLING	NONE	YES

Table 3. Comparison of Display Capabilities

Implementation

The initial driver development required understanding the principle data structures and calling conventions from the device independent Core to the driver itself. The device independent Core routines pass an "information packet" across the device independent/device dependent (DI/DD) interface which provides all operation codes and data values to the device driver. The calling conventions are further detailed in Appendix I of this document, and the GWCORE Design Document [36:20]. The information is a large, linear array (5000 32-bit words) that contains an operation code (opcode) with its associated integer and real data values. The array is named PARRAY and its contents are used by the device driver modules to produce the instructions for the particular device.

The modules make use of two data structures to control the state of the display and output instructions. A FORTRAN common block is used to hold current device parameters (for the ISC driver this common block is named ISCSAVE). The parameters include both device status variables and Core attribute values. A second common block (ISCBUFF) maintains an array that is used as the buffer for output to the display. The implementation of the driver required adding new common block variables for the new capabilities, adding the routines that update the new variables, and the addition of one module that outputs instructions for polygon filling. When graphics primitives are output to the screen, the common block values are inserted into the buffer along with data values and output to the device. The device driver performs the following sequence of steps to achieve the desired action.

General Device Driver Process

1. DI dispatcher passes "information array" to device driver.

2. Device driver performs a multiway branch to a driver functional module.

3. The functional module extracts values from the array.
   a) Value used to update parameters in device common block,
   b) or values are data used in plotting output primitives on display.

4. Control returns to the device independent sections of the Core.

As was previously stated, the Tektronix and ISC drivers were made as similar as possible. The implementation was divided into modules with the first modules patterned after the Tektronix driver. These modules included the straight-forward graphics functions for moves, and vector drawing as well as housekeeping functions like erasing the screen, controlling the cursor, and switching from graphics to alpha-numeric display. Subsequent increments in the display capability were provided by new modules which were then individually incorporated and tested. These modules included foreground and background color selection, character size selection, highlighting selection and filling polygonal areas.

Testing

Bottom-up testing was used during development of the individual modules. Because of the clean interface between the DI/DD sections of the Core, the calls from the DI Core to the DD software under development were made with a single, simulated controlling module or "driver". For testing purposes the values in the information array (PARRAY) were set in the pseudo-controlling module and passed through the dispatch emulator to the driver where individual modules were tested.

Once the modules were satisfactorily tested in bottom-up fashion with a pseudo-controlling module, the driver was incorporated with the Core package for integrated testing. Again, individual modules were tested and debugged followed by combining functions within larger test programs. All test programs were FORTRAN code that was compiled and linked with the graphics library and then executed. The test cases were designed to incrementally test the additional modules of the driver. Once an individual module was functioning, it was further tested by developing another "mini-application" program that exercised several modules concurrently to insure no side-effects had been introduced by the new module. The ISC driver is further described in Appendix A.

Input Routines

The input routines specified for the Core are, like the output capabilities, very extensive. They are, however, broken down into two areas to support synchronous and asynchronous input. The asynchronous functions require event-queue management and control over enabling, associating, and selecting the various possible input devices. On the other hand, the synchronous input functions provide the same services without the event queue and numerous device control requirements. The synchronous input functions sacrifice the flexibility of allowing the user greater input options at any instant for easier implementation. In addition to the simplified implementation requirements, the current hardware in the AFIT lab doesn't allow more than one input operation at a time. Therefore, only the synchronous input routines were considered. A subset of the logical input functions from section 6.2.9 of the Core were implemented.

Implementation

The starting point for development was Curling's design. In spite of Curling's observation that only synchronous input was possible with the hardware set-up of AFIT's lab [7:20], Curling provided more than necessary to satisfy the input routines and controls in some areas and insufficient design in other areas. The initialization process from Curling's design was implemented first. This required modifying the initialization of the Core to include setting the default values for the six input devices and the associated global common blocks. The input functions were then developed, as with device driver development, one function at a time. Each input routine required two significant developments before it could be tested. The device-independent routine and the corresponding device-dependent functions were developed together since they are different sides of the same interface. The device independent routines are all similar regardless of function. All of the input routines have an entry-level module and perhaps one or more secondary modules. Input routines that exercise the driver fill the information array with appropriate values and then call the dispatch routine.

Figure 6 depicts the module hierarchy of the implementation of the device-independent input routines and their use with the Tektronix

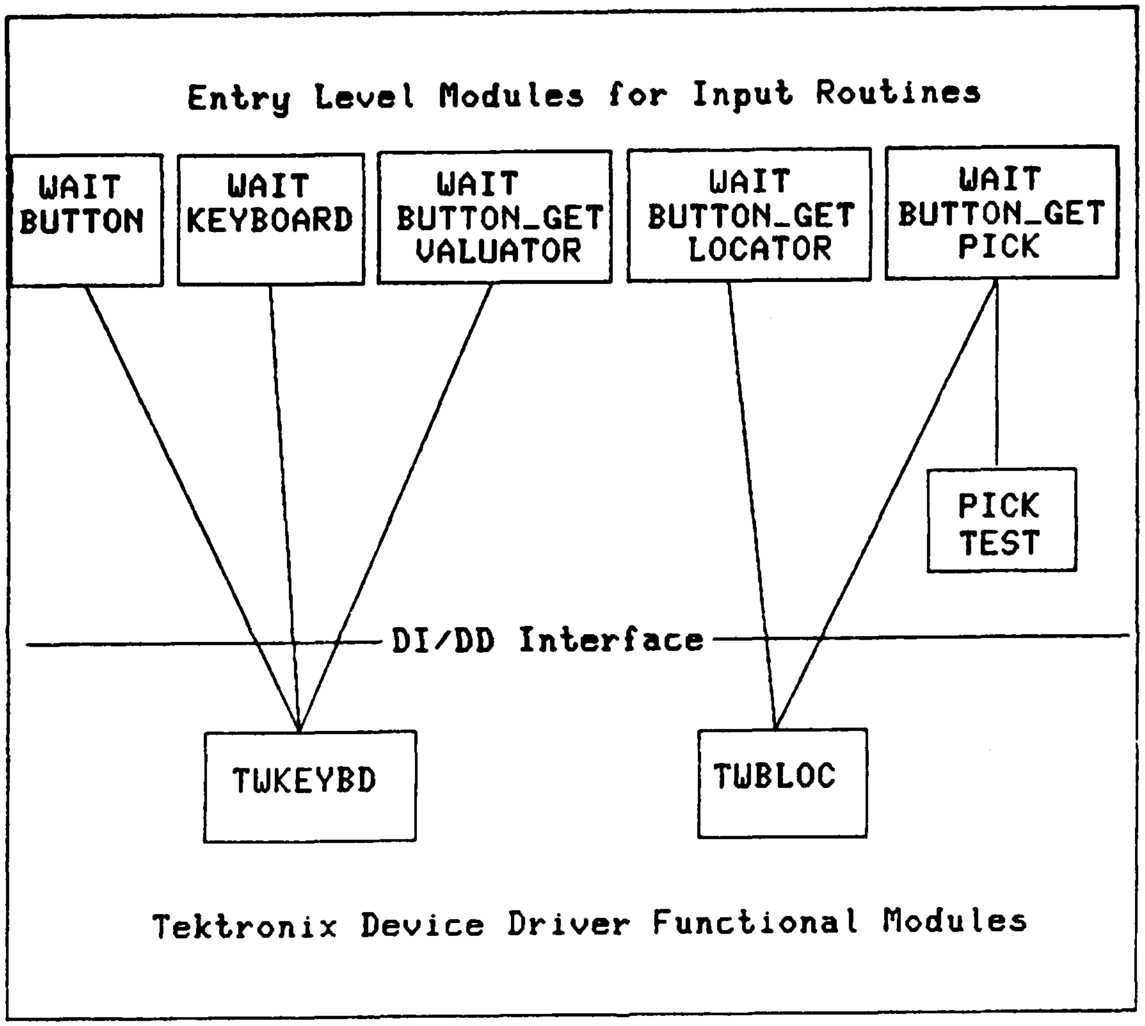


Figure 6. Module Hierarchy for Synchronous Input

device driver input routines. Note that this figure omits the modules that actually form the DI/DD interface. The dispatch routines and entry level module to the driver are not shown in order to more clearly show the relationship of the device-independent modules to the device-dependent driver modules for the Tektronix driver. This relationship would be different for another driver depending on the input devices available for a particular work station. The input routines perform the following sequence of steps to achieve the desired action.

<u>General Input Routine Process</u>

1. Call from application program to entry level module.

2. Request checked for validity.
   a) Core initialized at proper input and output levels.
   b) Parameters sufficient in type and number.

3. If valid then obtain appropriate instructions and values and insert them in 'informaton packet' to be passed across DI/DD interface to driver.

4. If invalid request then pass name of routine and invalid data or parameters to error reporting routine.

5. Return control to application program.

The development of the input routines required modifying the initialization to check that the specified input level can be supported and modifying the drivers to query the terminal to obtain its status. The device driver functions according to the particular input device's requirements and is independent of a particular operating system except for the input and ouput routines. The "wait-for-input" required for the synchronous input routines makes use of VAX/VMS operating system input/output service routines that allow specifying a timeout period after queuing a request to accept input on a particular channel. The input routines, as required by the Core standard, pass the timeout value as a parameter for its ultimate use by the driver in a call to a system service routine.

As was previously stated in chapter 3, the AFIT graphics environment has a minimal amount of input devices. In fact, the only input devices are the thumbwheels on the Tektronix terminal and the keyboard. Therefore, the logical input devices require simulation in order to meet the Core requirements for input devices. As developed, the device independent routines within the Core remain independent of the Tektronix device. On the device-dependent side of the Core, the Tektronix input driver modules use only two physical devices to accomplish the interactive input. The requirements for input devices and their implementation are summarized in Table 4. The ISC device driver was modified to support the keyboard and button functions in a manner similar to the Tektronix, however, the ISC terminal and keyboard have a very limited capability to support the other input primitives. The remainder of the discussion concerning the implementation of the input routines is for the Tektronix driver only.

CORE SYNCHRONOUS INPUT REQUIREMENT	DEVICE-INDEPENDENT IMPLEMENTATION	DEVICE-DEPENDENT IMPLEMENTATION
AWAIT_ANY_BUTTON	WAIT_BUTTON	WAIT_KEYBOARD
AWAIT_KEYBOARD	WAIT_KEYBOARD	WAIT_KEYBOARD
AWAIT_ANY_BUTTON_GET_VALUATOR	WAIT_VALUATOR	WAIT_KEYBOARD
AWAIT_ANY_BUTTON_GET_LOCATOR	WAIT_LOCATOR	WAIT_BUTTON_GET_LOCATOR
AWAIT_ANY_BUTTON_GET_PICK	WAIT_PICK	WAIT_BUTTON_GET_LOCATOR

Table 4. Requirements and Implementation for the Core Synchronous Input Routines

Because the DD side of the interface uses only the await-keyboard and await_any_button_get_locator subroutines these were developed first. The keyboard function in the device-independent part of the Core simply inserts the timeout value specified in the application program and the device number for the keyboard in the information array (PARRAY). Note that the device number is included solely for consistency with the asynchronous routines should they ever be implemented; in this case the device number is ignored. The device dependent part of the Core, namely the device driver, maintains a buffer to accept ASCII codes as they are returned from the system service call for input. This buffer allows input up to 8Ø characters in length and allows editing (deletions), too. When the input character is a carriage return the device driver returns the string and number of characters to the device independent routine. The DI routine then passes the values back to the application program. The await_button function is identical to await_keyboard except the keyboard buffer size is set to one so the first button pushed is returned to the application program.

The locator subroutine requires both input and output system service calls; the device driver output buffer is used to set the Tektronix terminal to Graphics Input mode. In this mode the graphics crosshairs and thumbwheels are enabled. Pressing any key transmits the horizontal and vertical cursor position in device units along with the key pressed, to the input buffer. The driver then converts the device units to normalized device units and passes these values and the character to the DI part of the Core and then to the application program.

The pick function is the most elaborate because the hardware support is minimal. The pick routine queries the terminal for the graphics cursor position and receives an (x,y) pair. This point must be converted to a segment name since all manipulations of display file objects is done with the segment name. Curling presumed the output-only Core routines would provide this function, however the GWCORE output routines required modification to support the pick function.

The algorithms to support the pick function use the "screen extent" of the figure. The screen extent is the smallest rectangle that completely encloses the figure. These screen extents are computed for each segment and stored in a table sorted by ascending order of screen extent. This table is created and maintained during the figure creation process. When a pick is requested by the application program, an (x,y) pair is returned by the driver. These (x,y) values are then compared with the extent values to determine if the point lies within an extent for a particular segment. If the point lies within the extent then that segment is deemed the "picked" segment. The disadvantage to this method is that a point may lie in more than one extent and can be confusing to an unwary user. The pick function will return the segment number of the smallest extent that contains the point, so, unless the picture is unusually complicated, there is probably a unique point that can be picked for the desired segment.

This is the only input function that is performed entirely independent of any driver. That is, all of the pick software resides on the DI side of the interface. This could have been put in the software for individual drivers but for the display devices available, i.e. the

Tektronix, the ISC, and the Avionics Lab's Ramtek would require this form of pick function execution. It therefore seemed appropriate to place the pick algorithms entirely in the device independent software so any display has this method available.

The valuator function uses the keyboard to input a real number. The Core specification for the functional capability of the valuator input primitive requires the input value to be compared with a range specified when the valuator function is initialized or set. For this implementation, the Set_Valuator function (section 6.2.11.8 of the Core) was not implemented but the range checking is still performed by the valuator function. The range is arbitrarily set at initialization of the input devices to a low of -5Ø.Ø and a high of +1ØØ.Ø. The high and low range values are maintained in the valuator common block.

The logical stroke input was not implemented but, could be accomplished with repeated calls to wait_button_get_locator and echoed by drawing connected lines between each point. This method for stroke input is tedious and for that reason Curling omitted the stroke function from his design. However, the implementation for the Tektronix would be straight forward since it means only repeated calls to previously developed input modules.

The lowest level modules perform the input/ouput operations necessary to interface the terminal with the Core software. Initially, the timed read functions required for each input module in the driver were included in the modules. However, for consistency with the output routine, calls to the system I/O services have been isolated in two modules READL and WRITL. The timeout value is used by READL when the SYS$QIOW system service is invoked to force the return from the input routine if no input from the user occurs. The Core specifies the values for the variables to be returned when a timeout occurs.

As was previously stated, a subset of the synchronous input routines were developed. The functions that were omitted are the following:

1. Await_Stroke_2
2. Await_Stroke_3
3. Await_Any_Button_Get_Locator_3.

These capabilities, along with the more extensive collection of device enabling/disabling, associating, and parameter setting routines were not implemented for two reasons. The input devices available in the AFIT environment cannot support these functions. Secondly, the input routines have been developed by George Washington University and should soon be available at AFIT. Therefore, the input routines were developed to make the best available use of the existing resources without spending too much time on code that may be replaced.

Testing

The initial testing of the individual input functions was performed bottom-up again with a simulated call to the driver to check for correct branching and parameter interfaces. After initial individual module testing, integrated testing was performed by developing an application program called ED, as in editor. The functions of a graphics editor were briefly discussed in chapter 3 and the interactive operation seemed an ideal means to develop test routines for the input devices and to experiment with the remaining objectives of integrating a suitable user interface in consonance with the tool design criteria of chapter 2 and 3. The most extensive testing of the input routines was accomplished during development of the application program.

Applications Development

The final development was intended to utilize all of Core input routines and demonstrate the applicability of integrating existing tools such as the HELP utilities and overall user-interface design. The application program is called INGRED for INteractive GRaphics EDitor. The editor provided an easy means to interactively develop pictures, test new input routines, develop and test modules for additions to the figure repetoire, develop a help library for easy response to help commands and develop the user friendly interface.

Implementation

There are four main sections to INGRED: the mainline and the three functional areas for FIGURES, ATTRIBUTES, and CONTROL. The module hierarchy for the interactive graphics editor is shown in Figure 7. The initialization of the editor takes place as the first step of the main part of the program. This involves initializing the default values, and displaying the graphics positioning aid (a grid) and the prompts. The mainline of the program is a loop that waits for keyboard input of a command, invokes the appropriate subroutines, and returns to waiting for the next command. The loop is terminated when the QUIT command is entered.

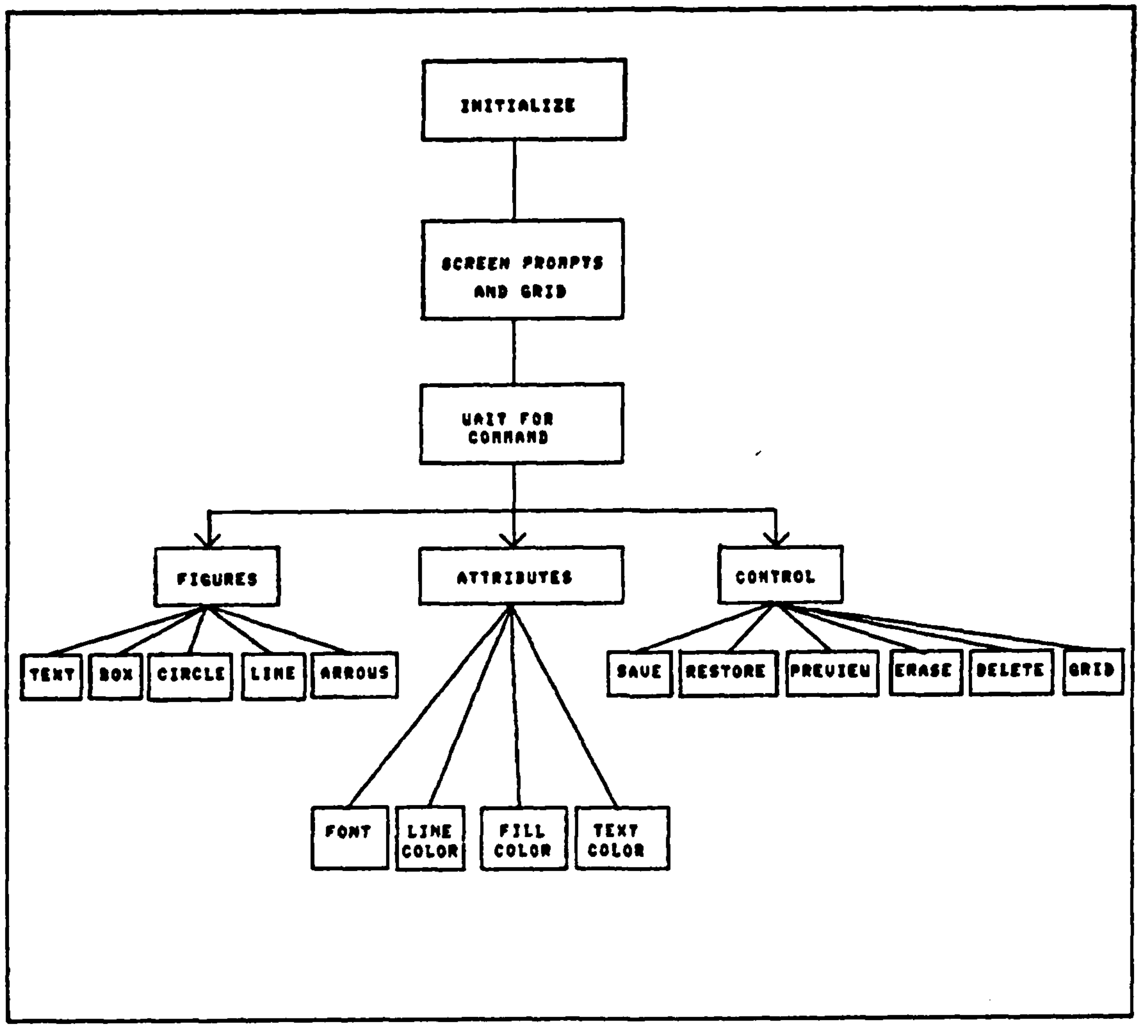


Figure 7. Module Hierarchy for the Interactive Graphics Editor Command Level

The bulk of functional modules for figure drawing were developed for the testing of the interactive input routines. The figures available to be drawn are quite simple and integration of additional figures could be easily accomplished. The rudimentary figures include a line, box, circle, arrowhead, and text. These objects are positioned on the screen at user designated points. The user selects a figure by typing in the figure name. The editor then prompts for selecting a position by displaying the graphics crosshairs. The requirements from chapter 2 demand that a consistent interaction technique be used for input to avoid confusion or frustration on the part of the user. Therefore, most of the figures require two positioning selections and then the figure is drawn to provide immediate feedback to the user. Additional aspects of the requirements that are reflected in the interface apply to all of the interaction techniques employed and are discussed following the description of the attribute and control modules.

Attribute control is achieved with the button function. For example, the text on the Tektronix may be the default hardware character generator or one of the four fonts available in the software. The selection is accomplished via a single button push. Similarly, the color selection for text, lines, and polygon filling is input by pressing a number between 1 and 8. These values correspond to the Core color values for a fixed color device like the ISC display.

Control functions allow editing and housekeeping commands to be performed. For example, deleting a previously drawn figure requires that the figure be "picked" first. Other editing commands allow storing and retrieving pictures on secondary storage, previewing partially completed pictures and changing the grid style to a set of points. All commands to the editor are input at the keyboard (through the wait_keyboard function call).

The criteria for tool development and effective user interface design as developed in chapter 2 were included in the implementation of the graphics editor. Prompts are provided during attribute selection for color and font selection. Other prompts are designed to prevent inadvertant deletion or modification of a picture by prompting before erasing the current picture in response to the ERASE or RESTORE command. The prompts appear after either command is entered to force the user to provide confirmation of a command with "significant" effects. Echoing is a requirement of the Core specification as well as user-interface design. All figures are echoed by drawing them on the screen as soon as their position is specified. Segments selected for deletion are echoed by drawing them again, then prompting the user to type yes or no in response to the question "Delete this segment?" Also, the user is allowed to abort an operation once it is selected. For example, if after entering the BOX (draw a box) command the user decides some other object is necessary then the function may be aborted by pressing the "A" key. The abort mechanism may be used any time the graphics crosshair is visible and the user will be returned to the command level.

Assistance to the beginning user is available through the HELP command. This command makes use of a VAX/VMS utility program that accesses a help library, extracts the desired topic, and prints it at the screen. The help library is a text library that is very similar to documentation on the subject. In fact, Appendix B is actually the text of the Help library. This method should allow the "shelf" documentation to correspond with the software action since the documentation is a printed version of the text available on-line. Hopefully, the help files will be modified when the source code is modified. To encourage this, the source code in the editor contains references (as comments) to the appropriate help files.

The final editor facility to be discussed allows the picture to be saved to disk and restored for further editing. Figure 5 showed the pseudo-display file as the device-independent input to the device driver. Saving the pseudo-display file on disk allows the device-independent picture instructions to be stored. This device independent picture description is called a metafile [1:175]. Restoring the picture involves reading the instructions back into the pseudo-display file and then invoking the device drivers. Note that further editing the picture implies reconstructing another data structure called the the segment table and associated values which point to the beginning and end of the segments in the display file.

The retrieval subroutine was extracted from INGRED and used in a separate application program called META. META allows finished pictures created and saved by INGRED to be displayed on either the Tektronix or ISC displays. The META program does not allow the picture to be edited but it still requires that the segment table be reconstructed so the function that causes the picture to be redrawn (NEW_FRAME) will work properly. The 'new frame' request loops through the segment table, i.e. the pointer variables that point to the beginning and end of segments in the display file, to redraw the picture. The META program could be used at different sites to draw pictures created elsewhere with GWCORE. META could also be incorporated in other application programs to retrieve and display any pictures created by GWCORE. Figure 8 shows the relationship between the two application programs and the picture generation and display capabilities in the AFIT environment.

Testing and Use

The testing of the interactive editor was done in stages that correspond to testing each of the functional areas. The initialization and mainline were implemented in skeleton form to establish a foundation for the addition of figures, attribute selection, and control modules. The figures are collections of primitives and are developed and tested as stand-alone graphics programs. This program is then inserted into the editor as a subroutine. The arguments for the figure (endpoints for lines, center and radius for circles, etc.) are passed from the interactive input routines. The mainline is modified to include a command corresponding to the newly added routine and the editor is then run to test the new module. In this way, additional figures are tested and subsequently added to the applications level of the graphics environment. The attribute routines are developed, tested and incorporated in the editor in a similar fashion.

Once there was a substantial collection of options for things like color indices, figures, and fonts it became difficult to remember the commands and their meaning. Therefore the prompts and error handling were added. The testing for these functions involved determining the best layout for the prompts and error messages. Because the Tektronix

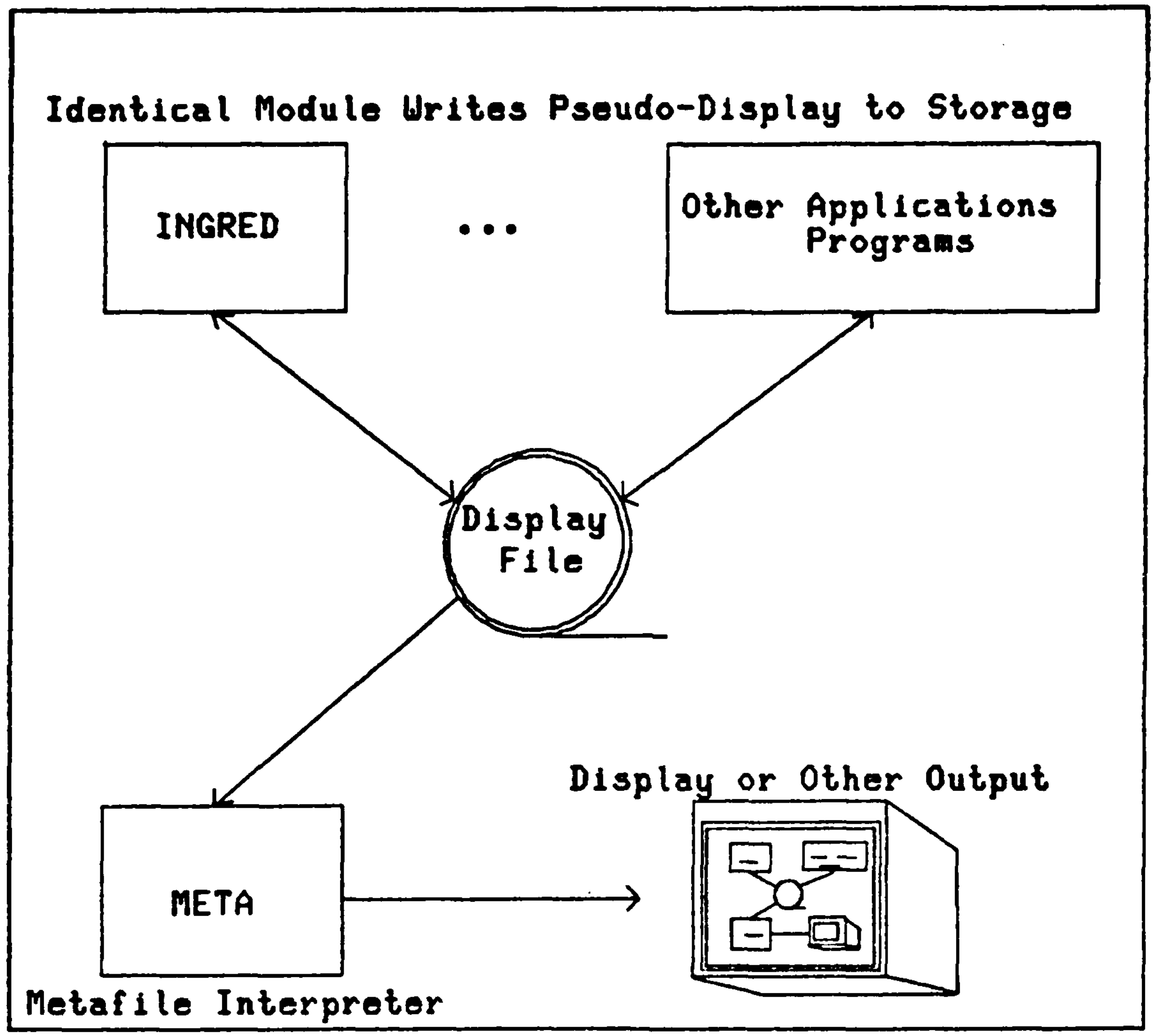

Identical Module Writes Pseudo-Display to Storage
INGRED
...
Other Applications Programs
Display File
Display or Other Output
META
Metafile Interpreter

Figure 8. INGRED and META are Tools for Use in the AFIT Graphics Environment

does not have selective erasure, the prompting has to be displayed at different locations on the screen to avoid the possibility of overwriting a previous message.

The best test of an applications program is to provide it to the users. I used the editor to prepare presentations as well as to test different capabilities. This testing pointed out some areas that might be modified after some use in a "production" environment. For example, the mainline automatically supplies segment names whenever a figure is drawn. However, naming the segments could be explicitly provided by the user to allow logical grouping of the figures. Segment naming may be a function of the editor that should be transparent for some users or explicit for others. Use of the editor on a regular basis might allow a preference to develop that could be subsequently incorporated in the editor.

Summary

Three major developments were described. The addition of a device driver for a color raster display for GWCORE was described. The synchronous input routines as specified by the Core standard were designed and implemented. Both the device-independent and device-dependent routines were developed; the device-dependent routines were implemented for the Tektronix device driver. The implementation of the color device driver and input routines were utilized by an application program that generates simple two-dimensional device-independent pictures. This interactive graphics editor was designed as a tool for preparation of simple figures.

V. Conclusions and Recommendations

The preceding chapters have documented the requirements, design strategy, and development of a general purpose graphics system for the VAX 11/780 at AFIT's Digital Engineering Laboratory. This chapter presents a summary of the this investigation's accomplishments as they pertain to the objectives presented in the earlier chapters. Recommendations are also furnished for further work in this area.

Conclusions

Much of the initial investigative effort went towards determining a strategy for development of the general purpose graphics system. Development of the graphics system using criteria derived from studies of software tool construction, user-machine interface design, and graphics environments were set as the goals. Software tool design requires that portability, modularity and utility be primary considerations for tool development. The user-machine interface requires facilities for consistent interaction and responses, prompting, echoing input, help (if necessary), and graceful error handling. The graphics system must have displays, input and output devices, and software for both graphics functions and device handling.

The proposed graphics system software standard was shown to include most of the requirements as derived from all three of the areas considered. The Core, as it is known, was developed with portability as a primary concern and specified the graphics capabilities for display and control needed for the graphics workstation. Any application developed with the Core should provide the effective user interface.

An implementation of the Core was extended to include the minimum input routines specified for the input devices. The synchronous input primitives were developed for the devices available. The logical input primitives for button, keyboard, pick, valuator and locator devices were produced for the Tektronix display. Since this terminal has only two physical input devices, namely the keyboard and thumbwheels, the logical devices for button, pick and valuator input were simulated in software. These simulations, however, are implemented in the device-dependent driver for the Tektronix so the device-independent routines remain applicable for future additions to the collection of input devices.

The specification of the graphics system provides more than adequate flexibility for input. The synchronous input routines supply all the functional capabilities needed for use with the current hardware and are significantly less complicated than the asynchronous input routines. In fact, another developer of a more extensive set of input routines suggests that the Core is over-specified [40], and I agree. In some cases the required functions simply cannot be performed with the given hardware. For example, the synchronous input routines require the ability to enable and disable the echo. However, there is no way to disable the echo facility (crosshairs) of the Tektronix 4014 and still place the device in the graphics input mode. For this reason, the requirements to set device characteristics were omitted. Similarly, due to the inadequacy of the input devices, the stroke input routines were not developed. The stroke input primitive is clumsy when the physical input device is a pair of orthogonal thumbwheels. However, the implementation of the input routines did provide for later addition of the capabilities if needed.

The other significant addition to the George Washington University version of the Core required writing a driver for the ISC 8001 color display. This proved somewhat difficult due to the lack of documentation for driver design and implementation. Even though GWCORE came with a driver for the Tektronix, this software was documented only through the sparse distribution of comments. Considerable effort went into deciphering the mechanisms used to pass data from the device-independent code to the device-dependent code. Hopefully, Appendix A and the source code for the ISC driver will provide sufficient information for further development. Note also the ISC display does not meet the requirements of Chapter 2 for device resolution. It was stated there that the lower limit on resolution should be 40 points per inch while the ISC is closer to 15 to 20 points per inch. This display is barely adequate for presenting bar charts.

Once the graphics library was complete with input routines and two display drivers, an application was developed. The Interactive Graphics Editor (INGRED) was developed not only to test the input routines but the ability to generate device-independent pictures, and to demonstrate the capability of the environment to support an effective user interface.

The device-independence of the graphics software was demonstrated by reproducing one picture file on either display. This capability is especially useful where some of the terminals have limited capabilities for interactive work, i.e., in an environment where only one workstation has input devices, the reproduction of interactively produced pictures is not limited to that particular display. Furthermore, the ability to save a picture file allows pictures to be transported between facilities with the same Core software. If the different facilities have other types of displays then the transferrability of picture files independent of the applications program allows effective use of the graphics resources.

The utility of the editor was demonstrated by preparing a variety of pictures for display and discussion. The editor was also used to prepare the tables and figures in this thesis. I believe the graphics editor would have been used more extensively if a hardcopy device had been readily available. The hardcopy for the figures and tables in this thesis were obtained with the Avionics Laboratory Tektronix display and hardcopy device. The production of figures and tables for theses and production of view-graphs for oral presentations is a task common to most AFIT students. However, as stated in Chapter 3, an application program is used only if it is perceived as being useful to the user. Unfortunately, it is difficult to sell the idea of using a graphics editor for picture production when no hardcopy capability is available.

The idea of developing tools as small, functional modules that can be combined to produce larger programs proved useful in developing the interactive editor and the program that reads the picture file from secondary storage for display. META is the metafile translator that reads a picture file back into the Core graphics software and then redraws the picture on a display. The editor required the capability to save a picture to disk (or tape) so that the picture could be retrieved for subsequent editing or display. The code for the retrieval capability in the editor was extracted and with little modification became META. META functions as a stand alone program but could be incorporated in an application program that requires retrieving picture files. Similarly, the code to save a picture file on disk is a small subroutine in INGRED and could easily be inserted in other applications that require saving picture files.

Recommendations

Areas for further development can be divided into two categories: additional capabilities could be added to the graphics environment and to the area of applications programs.

The graphics hardware environment could be greatly improved by adding a more modern graphics workstation. This should include a high-resolution (greater than 512 X 512 points) raster-scan, color display with input devices including button devices, a tablet and a pick device like a joystick or light-pen. The Evans & Sutherland workstation should improve the capabilities significantly, especially if the input devices available for the terminal are included. One of the primary deficiencies that should be overcome is in the area of hardcopy output. Drivers are needed for the existing Calcomp drum plotter and Printronix dot-matrix printer. The color display will require a color hardcopy capability too.

The graphics software environment could be extended in many areas by adding capabilities for hidden-line and hidden-surface removal, additional input routines associated with a new workstation, and the more current research areas in realistic scene development with complex shading, shadowing and transparency.

As was mentioned in chapter 3, the proposed international graphics standard, GKS, is receiving more and more attention. The commercial market for graphics has supported the Core proposal and many companies are either adding GKS compatibility to their Core implementations or are altering their plans to produce and support the Core [33]. Another possibility for software development would be to obtain the specification of GKS and perhaps an implementation, as was done for the Core proposal. An analysis of the advantages and disadvantages of the proposals might be very useful to prospective users and purchasers of one or the other system.

The second area for possible further work could involve applications development. Integrating the graphics and database capabilities of the VAX would allow computer-aided design capabilities to be developed. Alternatively, the existing database containing faculty and student data, classroom and class schedules could be accessed by an applications program to prepare standard business-type charts. These charts could portray management information on the number of students or faculty by rank or specialty area, room utilization, room schedules, budgets and so on. I believe the graphics system available on the VAX provides an excellent base for further growth.

Bibliography

1. Foley, James and Andries Van Dam. Fundamentals of Interactive Computer Graphics, Reading, Massachusetts: Addison-Wesley Publishing Company, 1982

2. Newman, William and Robert Sproull. Principles of Interactive Computer Graphics. (Second Edition) New York: McGraw-Hill Book Company, 1979

3. "Status Report of the Graphics Standards Planning Committee of ACM/SIGGRAPH," Computer Graphics, 13 (3): (August 1979)

4. Miller, Edward, editor. Tutorial: Automated Tools for Software Engineering. IEEE Computer Society, 1979

5. "National Bureau of Standards Programming Environment Workshop Report," ACM Software Engineering Notes, 6 (4): (August 1981)

6. Chappell, Gary. "Implementations of the CORE," Computer Graphics, 13 (4): 26Ø-278 (February 198Ø)

7. Curling, Harold. Design of an Interactive Input Graphics System Based on the ACM Core Standard, MS thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, December 198Ø. (AFIT/GCS/EE/8ØD-6).

8. Tarbell, Philip. Continued Development and Implementation of a Standard Graphics Package for the AFIT VAX 11/78Ø, MS thesis. Wright-Patterson AFB, Ohio: Air Force Institute of Technology, December 1981. (AFIT/GE/EE/81D-58).

9. Kernighan, Brian W. and P. J. Plauger, Softare Tocıs in Pascal, Reading, Massachusetts: Addison-Wesley Publishing Company, 1981

1Ø. Dolotta, Frank, and J. R. Mashey. "An Introduction to the Programmer's Wokbench," Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society, October 1976

11. Wegner, Peter. "The Ada Language and Environment," ACM SIGSOFT Software Engineering Notes, 5 (2): April 198Ø

12. Ingalls, Daniel H. H. "The Smalltalk Graphics Kernel," Byte, 6 (8): August 1981

13. Tesler, Larry. "The Smalltalk Environment," Byte, 6 (8): August 1981

14. Prentice, Dan. "An Analysis of Software Developcment Environments," ACM SIGSOFT Software Engineering Notes, 6 (5): October 1981

15. Foley, James D. and Victor L. Wallace. "The Art of Natural Graphic Man-Machine Conversation," Proceedings of the IEEE, 62 (4): April 1974

16. "GIGI Software," Software Product Description, Digital Equipment Corporation bulletin AE-L308B-TK, February 1982

17. Magnenant-Thalmann, Nadia, et. al. "GRAFEDIT: An Interactive General-Purpose Graphics Editor," Computers & Graphics, 6 (1): January 1982

18. Bono, Peter R., et. al. "GKS - The First Graphics Standard," IEEE Computer Graphics and Applications, 2 (5): July 1982

19. Lohr, James A. "Dedicated Graphics Computers Boost System Efficiency," Digital Design, 12 (2): February 1982

20. Mitze, Robert W. "The UNIX System as a Software Engineering Environment," Software Engineering Environments, Amsterdam: North-Holland Publishing Company, 1981

21. UNIX Programmer's Manual, Bell Telephone Laboratories, New Jersey 1979, Vol 2A, 7th edition

22. Carlson, Eric D. "Graphics Terminal Requirements for the 70's," Tutorial: Computer Graphics, IEEE Computer society, 1979

23. Straayer, David H. "Hoisting the Color Standard," Computer Design, 21 (7): July 1982

24. Lucido, Anthony P. "Software Systems for Computer Graphics," Tutorial: Computer Graphics, IEEE Computer Society, 1979

25. Langhorst, Fred E. "Working Towards Standards in Graphics," Computer Design, 21 (7): July 1982

26. Wasserman, Anthony J. "User Software Engineering and the Design of Interactive Systems," Proceedings of the Fifth International Conference on Software Engineering, IEEE Computer Society, 1981

27. Moynihan, John A. "What Users Want," Datamation, 28 (4): April 1982

28. "MOVIE.BYU - A General Purpose Computer Graphics System," Users Manuals for Version 3.7, Brigham Young University, Utah. January, 1980

29. "Color Graphics Language (CGL) Reference Manual," Users Manual for 6210 Color Graphic Computer Terminal, Ramtek Corporation, California, 1980

30. O'Hair, Kelly. "GRAFLIB Advanced User Manual," Lawrence Livermore National Laboratory, California, 1981

31. Hausen, Hans-Ludwig and Monika Mullerberg. "Conspectus of Software Engineering Environments," Proceedings of 5th International Conference on Software Engineering, March 9-12, 1981

32. "Graphics in Business," Datamation, 28 (9): August 1982

33. Shrelo, Kevin B., "ANSI Graphics Standards Coalesce Around International Kernel," Mini-Micro Systems, 15 (11): November 1982

34. Newman, William and Andries Van Dam. "Recent Efforts Towards Graphics Standardization," Computing Surveys, 10 (4): December 1978

35. Michener, James C. and Andries Van Dam. "A Functional Overview of the Core System with Glossary," Computing Surveys, 10 (4): December 1978

36. Wenner, Patricia, et. al. "Design Document for the George Washington University Implementation of the 1979 GSPC Core System," GWU-IIST-80-06, Washington, D.C., May 1980

37. "How to Use the 8001 CRT," Intelligent Systems Corporation, Georgia, 1978

38. "Precision Visuals Product Bulletin, DI-3000," Precision Visuals, Boulder, Colorado. January 1982

39. "User Document for the GWU Core System on the VAX 11/780," George Washington University, Washington D.C., February 1981

40. Hanlon, James. "Implementation and Evaluation of Input Functions for the 1979 Core Graphics System," M.S. Thesis, Electrical Engineering/Computer Science Department, George Washington University, Washington D.C., May 1981

Appendix A

Device Driver Development

This appendix describes the principle data structures and data flow used to drive a graphics device with the George Washington University implementation of the ACM SIGGRAPH Core Standard, known as GWCORE. This appendix should be used in conjunction with the Design Document [36] and replaces section 5.14 of that document.

The Core standard is designed to be as nearly independent of any graphics peripheral and operating system as possible to provide a measure of portability for the implementations. This is accomplished by isolating the functions that must, of necessity, be specific to a particular device or operating system service. The structure of the Core therefore requires device specific drivers be developed as a module separate from the Core and application program software.

The GWCORE structure contains a single interface between the Core viewing transformations that produce device-independent instructions for a picture and the device drivers that interpret the instructions for a specific graphics peripheral. The boundary is called the Device Independent/Device Dependent (hereinafter called DI/DD) interface. The next sections describe the data structures pertaining to this interface and the device drivers themselves. How the data is used by the driver is presented in the section on data flow.

DI/DD Common Data Structure

A single data structure contains all of the information that is passed between the device independent and device dependent routines. This information packet is an array of 5000 32-bit words and contains both REAL and INTEGER*4 data types. The array is called PARRAY and is equivalenced to an identical array of type REAL. In this way, one linear data structure can contain both real and integer values.

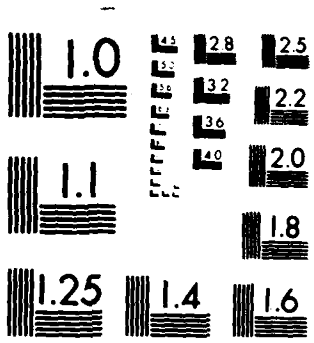
The contents of this array consists of one or more operation codes (opcodes) as assigned by the DI graphics routines. Each opcode has a fixed length and associated integer and real data values. There are approximately 70 functions each with its own opcode and data values. These functions constitute the device-independent instructions that are passed to the device driver. Execution of the opcodes is in a manner consistent with a particular device's graphics capabilities.

Some of the codes are similar in that they serve the same purpose however, their data values are different to accomodate various display device capabilities. For example, color attributes can be specified by direct index (using function 35) or by color and intensity values (functions 31 and 33). The ISC color display uses only function 35, while the Tektronix ignores all color instructions. The function codes are explained in section 3 of the Design Document. A data element description for the information packet follows.

AD-A124 694 DEVELOPMENT OF AN INTERACTIVE COMPUTER GRAPHICS SYSTEM LIBRARY AND GRAPHICS TOOLS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K W ROSE 2/2

UNCLASSIFIED DEC 82 AFIT/GE/EE/82D-58 F/G 9/2 NL

END
DATE
FILMED
3 84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

Data Element Description

Data Element Name: Common block PAR

Description: PAR is the name of the common block that contains the instructions and data values for transmission across the DI/DD interface.

Aliases: The common block is always the same: the elements PARRAY and RARRAY refer to the same data structure however.

Composition: Common block Par contains one heterogenous linear array containing both REAL and INTEGER*4 values. PARRAY is declared 5000 INTEGER*4 values; RARRAY is declared 5000 REAL values and the two are then assigned the same memory locations by using a FORTRAN equivalence statement.

Associated Processes: All device independent graphics subroutines access PARRAY and RARRAY through the global common block PAR. The PARRAY is passed to the device driver through the device dispatch subroutine, GDSPCH.

Associated Records: The contents of PARRAY are also stored in the device-independent display file, FILE.

Data Characteristics: The first four values are always as follows:

1) length of information array for this opcode (different depending
   on the opcode)
2) opcode
3) number of integer data values for this opcode
4) number of real data values for this opcode. The remainder of the array (element 5 through LENGTH) are the REAL and INTEGER values. The assignment of integer values to the data structure require using PARRAY as the array variable name while the assignment of real values to the data structure require using RARRAY as the array name.

Device Driver Data Structures

The entry level module to a device driver uses the values from PARRAY, as passed by the DI dispatching routine, to form integer and real arrays with values identical to the integer and real parameters for the opcode. The arrays, IARRAY and RARRAY, are then passed to the functional modules of the driver where they are used to update global values pertaining to device attributes or are used as data in drawing the picture. The global values are maintained in a FORTRAN common block and reflect the device parameters to be used for drawing pictures. Otherwise, the values are passed to the device directly as data, usually as screen coordinates.

Device Common Block

The ISC display driver, for example, uses common block ISCSAVE to hold current values for the Core attributes as set for the device. For example, the variables include values for line color, text color, and logical values indicating segment visibility settings. This common block also contains values that pertain to the device attributes, for instance the current normalized device display dimensions and a state variable indicating graphics or alphanumeric mode. The values in the common block are initialized to default values just after the Core is initialized; the Core specification requires initialization as the first call to any Core routine. The device driver is initialized when a surface is selected for output. The device values may be changed by the Core during execution of a graphics program when the DI routines send the appropriate opcode and data values.

The bulk of the modules within the device driver serve to update the common block variables for the attributes. However, of the 18 subroutine modules of the ISC device driver, four cause output of graphics primitives to the display. These four are modules for drawing text, markers, regular vectors and vectors used for polygon filling. The output of these primitives is preceded by inserting appropriate attribute values obtained from the the common block and commands specific to the device. For example, if a vector is to be drawn, then the plot instruction is preceded by commands to set the color to the current linecolor value before the plotting the vector. The data element description for the device common block as used in the ISC driver follows.

Data Element Description

Data Element Name: Common block ISCSAVE

Description: Device attributes for ISC 8001 color graphics display.

Aliases: None

Composition: ditype: device interface type (not used)
ifile: file pointer (not used)
marker: current marker character
linecol: ASCII value accepted by ISC as a color
textcol: ASCII value for text color setting
fillcol: ASCII value for polygon fill color
bcol: ASCII value for background color
pedge: polygon edge style (not used)
highlt: highlighting (blink) flag
chrsiz: character size flag
grafmd: graphics mode (or alpanumeric mode) flag
start: control flag
ivisib: graphics primitives visibility flag
ubatch: batch of updates flag
ndcxx: normalized display width (x)
ndcyy: normalized display height (y)
xorg: x origin of normalized display surface
yorg: y origin of normalized display surface
size: maximum of ndcxx or ndcyy
ix: x cursor location
iy: y cursor location

Associated Processes: All functional modules of the ISC driver acces values of ISCSAVE.

Associated Records: Values are computed or assigned to the variables based on the integer and real parameters passed across the DI/DD interface.

Data Characteristics: see composition.

Device Buffer

The output of graphics instructions to the graphics device is accomplished by transmitting ASCII control and escape codes in addition to data values. These instructions and data points are computed within the driver functional modules and inserted in a buffer prior to transmission. The ISC device uses an array of data types declared LOGICAL*1 which allows all commands and data values to be 8-bit bytes that can assume values in the range 0 to 255. This buffer is named MBUF and is in common block named ISCBUFF.

The values inserted in this buffer, as pointed out in the previous section, are both command values understood by the device and data values. The functional modules of the driver insert the values in the buffer by calling another subroutine whose sole purpose is inserting the values, updating a count of the values inserted, and maintaining a pointer to the next array position to be filled. The subroutine ISCPACK serves this purpose in the ISC device driver. This same subroutine calls the system service routine that sends the buffer if one of two conditions is met. If the counter in the pack routine exceeds the size of the buffer then the buffer is transmitted to the device. The alternative is to have a functional module of the driver call the pack routine with a value that corresponds to an instruction to flush the buffer. In the case of the ISC driver, ASCII value 250 decimal ('FA' in hexadecimal) causes ISCPACK to call the transmission routine.

The drivers for the Tektronix and ISC displays are similar, especially for the common blocks for the device attributes and output buffer. Therefore the data element descriptions are similar except for the ISC variables for color, highlighting, and character size. The data element description for the buffer follows.

Data Element Description

Data Element Name: ISCBUFF

Description: Buffer used to transmit ASCII values to control the display device.

Aliases: None

Associated Processes: All functional modules of the ISC driver may deposit values in the buffer. The buffer is filled by subroutine ISCPACK. ISCPACK will flush the buffer if additional insertions would cause overflow or ISCPACK receives a value to be inserted that is interpreted as a command to flush the buffer. Subroutine WRITL is called by ISCPACK to perform actual transmission of buffer to the display.

Data Characteristics: LOGICAL*1 types can assume values in the range Ø to 255; this corresponds to the ASCII code values. All assignments to the buffer use a convenient mnemonic. The values for the mnemonics are assigned by a FORTRAN DATA statement to a hexadecimal constant.

Figure 9 is a data structure diagram portraying the relationships of the data elements in the driver and the device independent part of the GWCORE package.

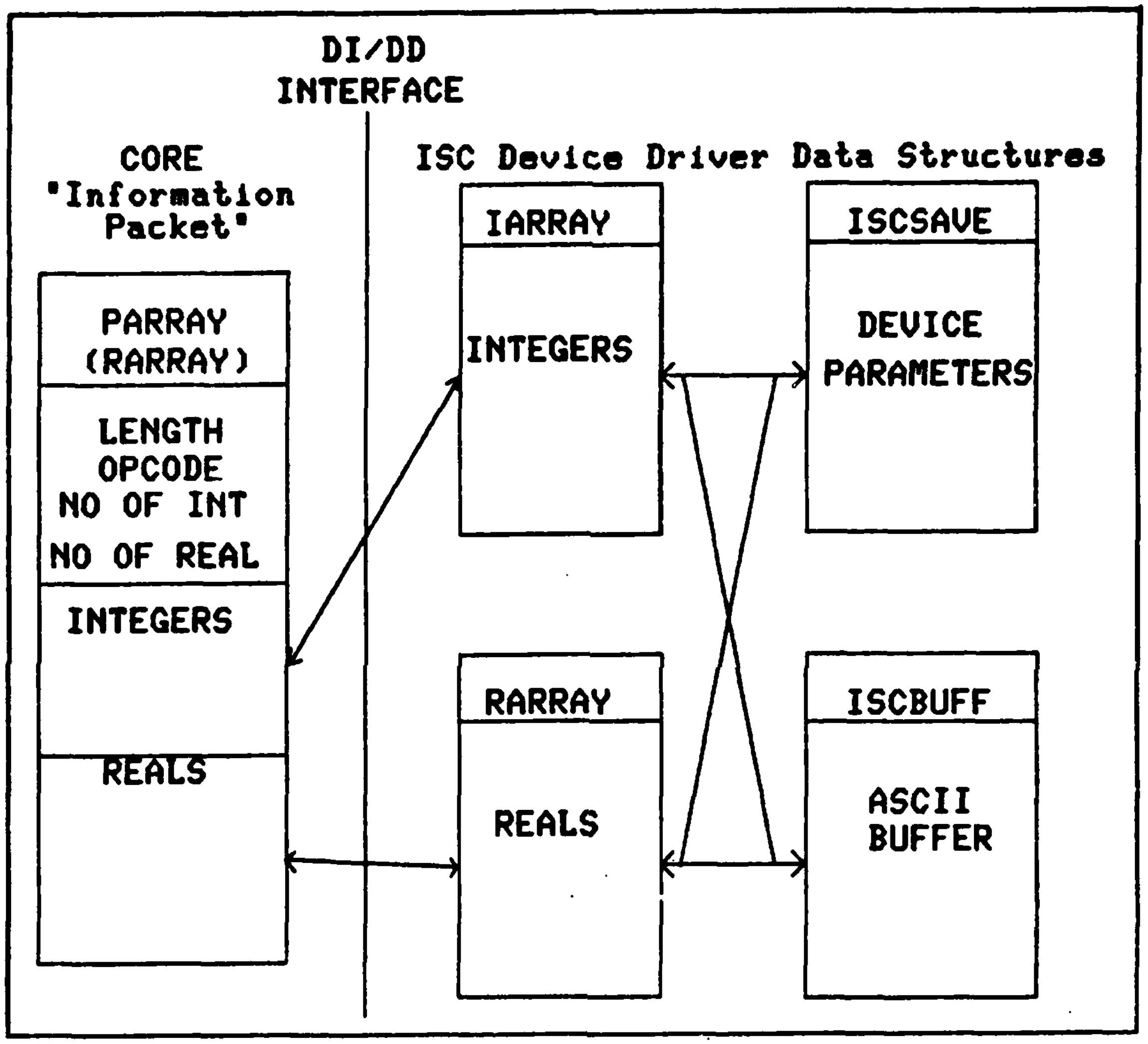


Figure 9. Data Structure Diagram for the Device Driver

Device Driver Data Flow

The driver is called from the DI dispatching routine and passed the values of the PARRAY that dictate the desired operation. (The dispatch calling parameter list can be found in the Design Document in section 2.5, page 21). The entry level module of the driver uses a multi-way branch on the opcode to a statement that calls the appropriate functional module. The call to the module then passes the appropriate integer and real array values.

As was stated in the preceding section, the modules either update device common block values or insert values in the buffer for transmission to the display device. The updating of common block values is straightforward but the output of the buffer to the display device requires some further explanation.

The buffer that is sent to the graphics device uses a low-level system service call, SYS$QIOW. This call to the operating system is VAX/VMS specific. All of the VAX/VMS specific routines have been isolated in three driver modules: WRITL (output), READL (input) and WAIT (synchronization delays). WRITL and READL use other system services to determine what output channel to assign to the I/O request, and assign a name to the channel. Also, arguments to the SYS$QIOW call allow ASCII control and escape codes normally intercepted by the operating system to be passed directly to the device.

The structure chart in Figure 10 depicts the organization of the modules in the device driver.

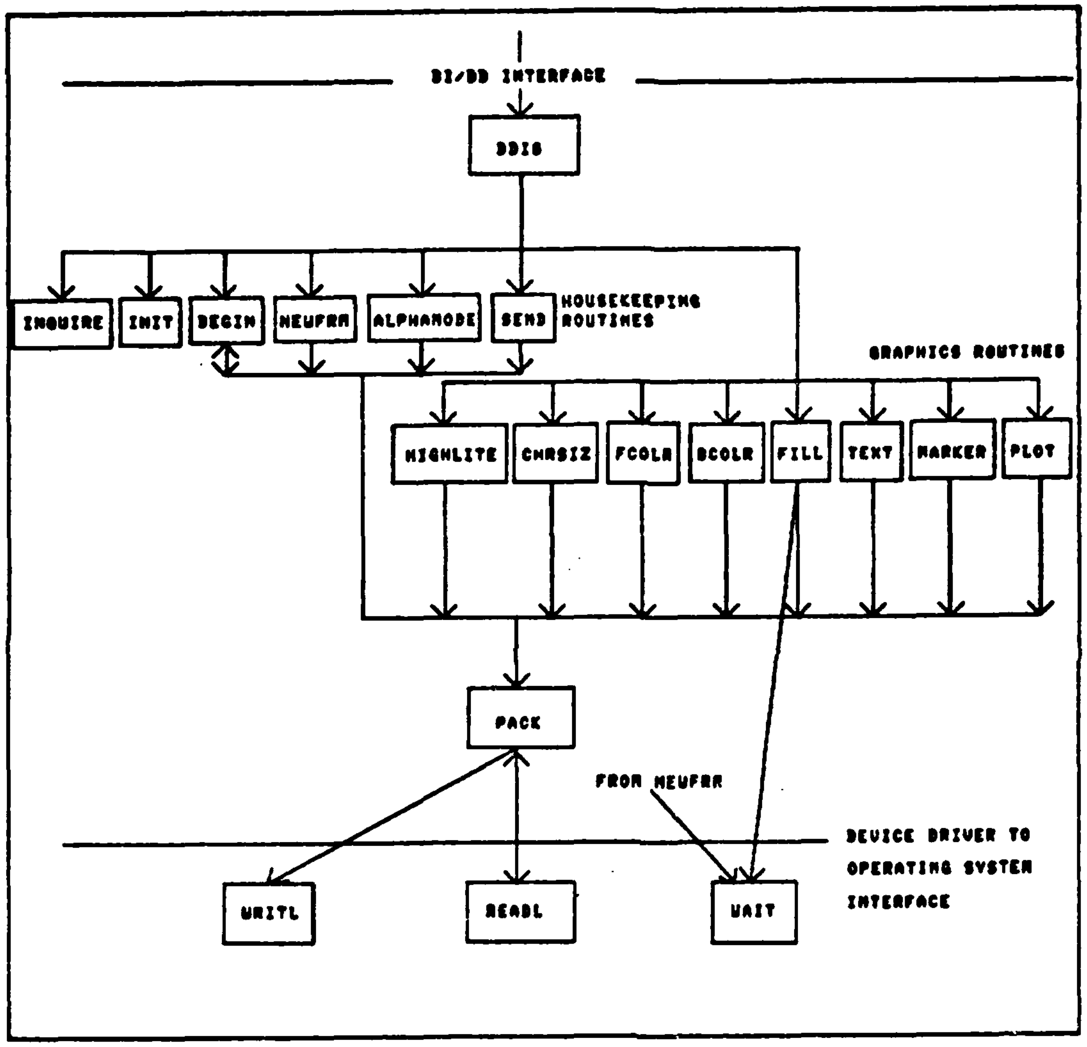


Figure 18. Module Hierarchy for the ISC Device Driver

Device Driver Routines for Input

The development of the input routines required wri*ing code for both the DI part of the CORE and in the device dependent code of the driver. The bulk of the input routines were written for use with the Tektronix display. The changes required inventing and adding the new opcode branch routines, developing the driver input functions and modifying the low-level system service calls to perform the timed input requests.

The opcodes were established first. The opcodes determine the sequence of values assigned to PARRAY in the DI part of the CORE then passed across the DI/DD interface. The opcodes and their associated parameters are listed below; the format matches the opcode description as found in the GWCORE Design Document. The function number corresponds to the opcode. The number of variables in the integer and real description correspond to the number of integers and reals transmitted (or returned) across the DI/DD interface. Recall that the first number in the "information packet" is the length of the packet for a particular opcode. This number can be computed by adding the number of integers plus the number of reals plus four (the preamble that provides the opcode and length values). This somewhat Byzantine method of describing the information packets is consistent with the GWCORE documentation and is best understood by examining a listing of a picture file produced by INGRED. The opcodes were arbitrarily assigned as follows.

```
Function 100  Wait_Any_Button_Get_Valuator
Integer Values:
     TIME,VALNUM ... timeout value and valuator number.
Real Values Returned:
     VAL ...  value.


Function 102: Wait_Button
Integer Values:
     TIME ... timeout value.
Integer Values Returned:
     BUTN ... (one ASCII value) button name.


Function 104: Wait_Keyboard
Integer Values:
     TIME, KEYNUM ... timeout value and keyboard number.
Integer Values Returned:
     LEN ... length of string,
     INSTR ... (up to 80 ASCII values) character string.


Function 106: Wait_Any_Button_Get_Locator_2
Integer Values:
     TIME, LOCNUM ... timeout value and locator number.
Integer Values Returned:
     BUTNUM ... button number (ASCII).
Real Values Returned:
     X,Y ... NDC coordinates of cursor location.
```

The parameter list and function names correspond to the User's Document [39]. The multiway branch in DDTX (the entry point to the Tektronix driver) passes the PARRAY values to the driver functional modules. The PARRAY values are passed in the integer array (IARRAY) and real array (RARRAY) as before. The functional modules in turn call the lower level system service call to queue an input request with a specified timeout period. The wait time is a user specified value in the entry level module of the device independent part of the code.

The integer and real arrays are the only means of passing the input back to the device independent routines for their use. For this reason, all character values are returned as integer values corresponding to the equivalent ASCII character code. The device independent input routines convert the values to a character string.

Appendix B

INGRED and META User's Manual

INGRED is an interactive graphics editor that allows creation and editing of two-dimensional figures. INGRED allows creation of the pictures on the Tektronix 4014 for display on any graphics device that has a suitable driver for the CORE graphics software library. The display capabilities are not limited to the 4014, for example, color may be specified and will appear on the ISC 8001 color display. Additional attributes are available as well; for text one of three software fonts may be selected (usable on the 4014 only.) Editing commands are available for saving and retrieving picture files, adding and deleting segments, erasing the screen to start again, and previewing the picture without the prompting and positioning aids.

META reads picture files in from secondary storage (usually disk although a tape reading capability could be easily added) and reproduces the picture on the selected display. META prompts for selecting a device (4014 or ISC), displays the picture, prompts again with 'OK' in the lower right corner of the picture when the picture is complete and awaits any keyboard input to prompt for another picture file name.

Both INGRED and META are on the AFIT VAX 11/780 disk pack with the volume label AFITUSER. To run either program, set the default directory to DMA1:[ROSE.TEST], then RUN INGRED or RUN META. Both INGRED and META use DMA1:[ROSE.PIKFILE] for storing and retrieving the picture files.

Using the Interactive Graphics Editor, INGRED

The following brief narrative describes the input and response of INGRED as shown in the accompanying hardcopy of the Tektronix display. Once the default directory is set as described above, typing RUN INGRED will initiate the interactive session. INGRED draws the grid that serves as a positioning aid and then lists the available commands down the left side of the grid. The 'COMMAND:' prompt appears at the bottom of the grid along with the flashing block cursor. At this point, INGRED expects the user to use the keyboard to input any of the commands listed within the boxes. As can be seen in the first example, two commands have already been typed, namely, 'FONT' and 'TEX.'

The first command, FONT, results in the secondary prompt, 'FONT TYPE? (1-4):' and the user is expected to input a single integer in the range indicated. Note that for commands, the command is entered by pressing the return key, while input for single integers like the FONT, require only pressing the key to return to the command level. The second command, TEX, is the abbreviated form of the TEXT command. Entering this command produces the graphics crosshair cursor. The crosshairs may be positioned with the thumbwheel controls on the Tektronix keyboard. The TEXT command expects the user to position the cursor at the lower left corner of the TEXT string that will be entered. Once the crosshairs are positioned, pressing any key will return the block cursor at the point previously marked by the graphics crosshairs. The user should then input the text string desired. The text is echoed (displayed) in the Tektronix font style. The RUBOUT key is used to delete erroneous input. The string is transmitted with the return key. INGRED then redraws the string with

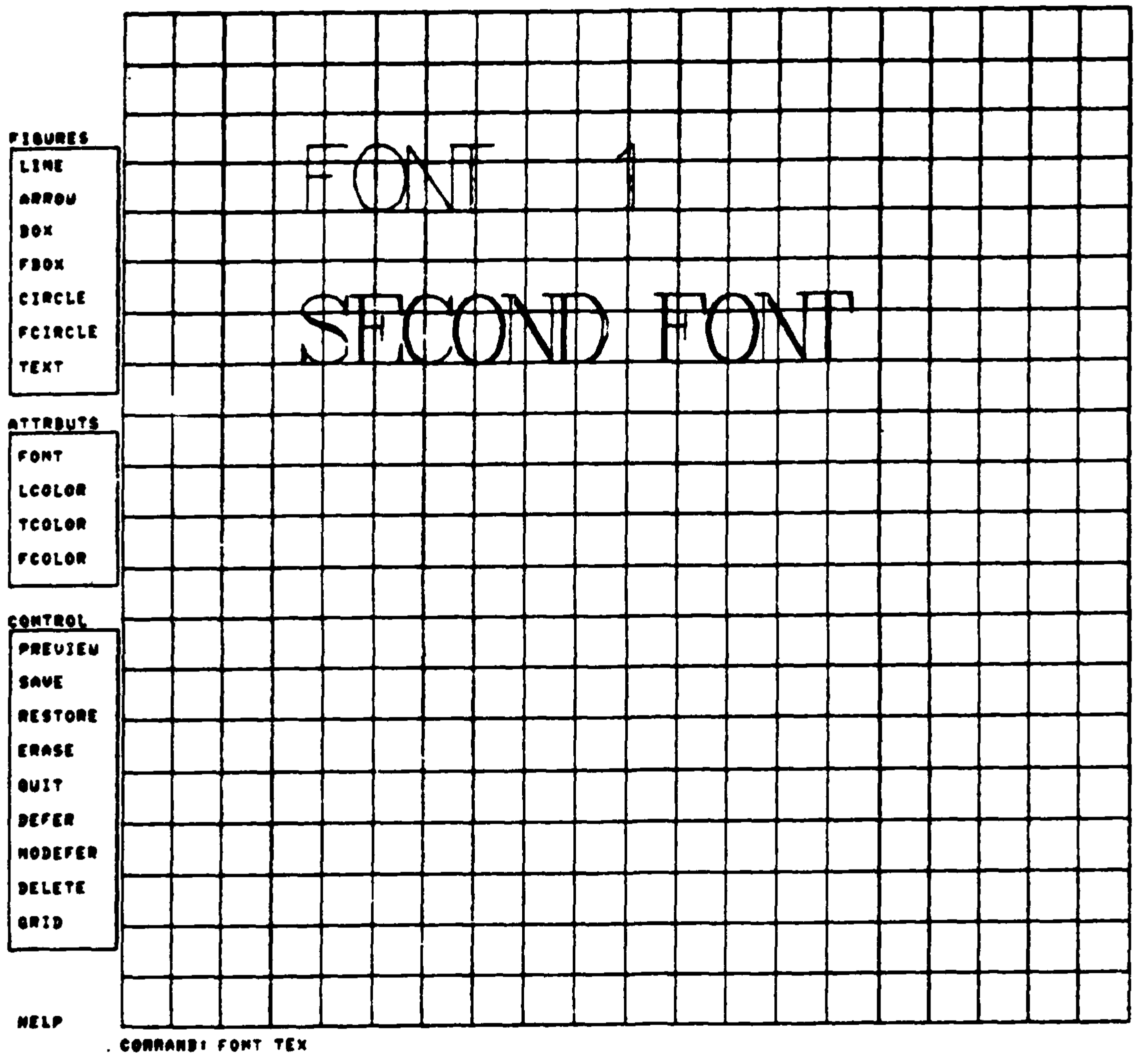
<u>Using the Interactive Graphics Editor, INGRED</u>

The following brief narrative describes the input and response of INGRED as shown in the accompanying hardcopy of the Tektronix display. Once the default directory is set as described above, typing RUN INGRED will initiate the interactive session. INGRED draws the grid that serves as a positioning aid and then lists the available commands down the left side of the grid. The 'COMMAND:' prompt appears at the bottom of the grid along with the flashing block cursor. At this point, INGRED expects the user to use the keyboard to input any of the commands listed within the boxes. As can be seen in the first example, two commands have already been typed, namely, 'FONT' and 'TEX.'

The first command, FONT, results in the secondary prompt, 'FONT TYPE? (1-4):' and the user is expected to input a single integer in the range indicated. Note that for commands, the command is entered by pressing the return key, while input for single integers like the FONT, require only pressing the key to return to the command level. The second command, TEX, is the abbreviated form of the TEXT command. Entering this command produces the graphics crosshair cursor. The crosshairs may be positioned with the thumbwheel controls on the Tektronix keyboard. The TEXT command expects the user to position the cursor at the lower left corner of the TEXT string that will be entered. Once the crosshairs are positioned, pressing any key will return the block cursor at the point previously marked by the graphics crosshairs. The user should then input the text string desired. The text is echoed (displayed) in the Tektronix font style. The RUBOUT key is used to delete erroneous input. The string is transmitted with the return key. INGRED then redraws the string with the current font selected. The first example illustrates two different fonts available in the graphics software.

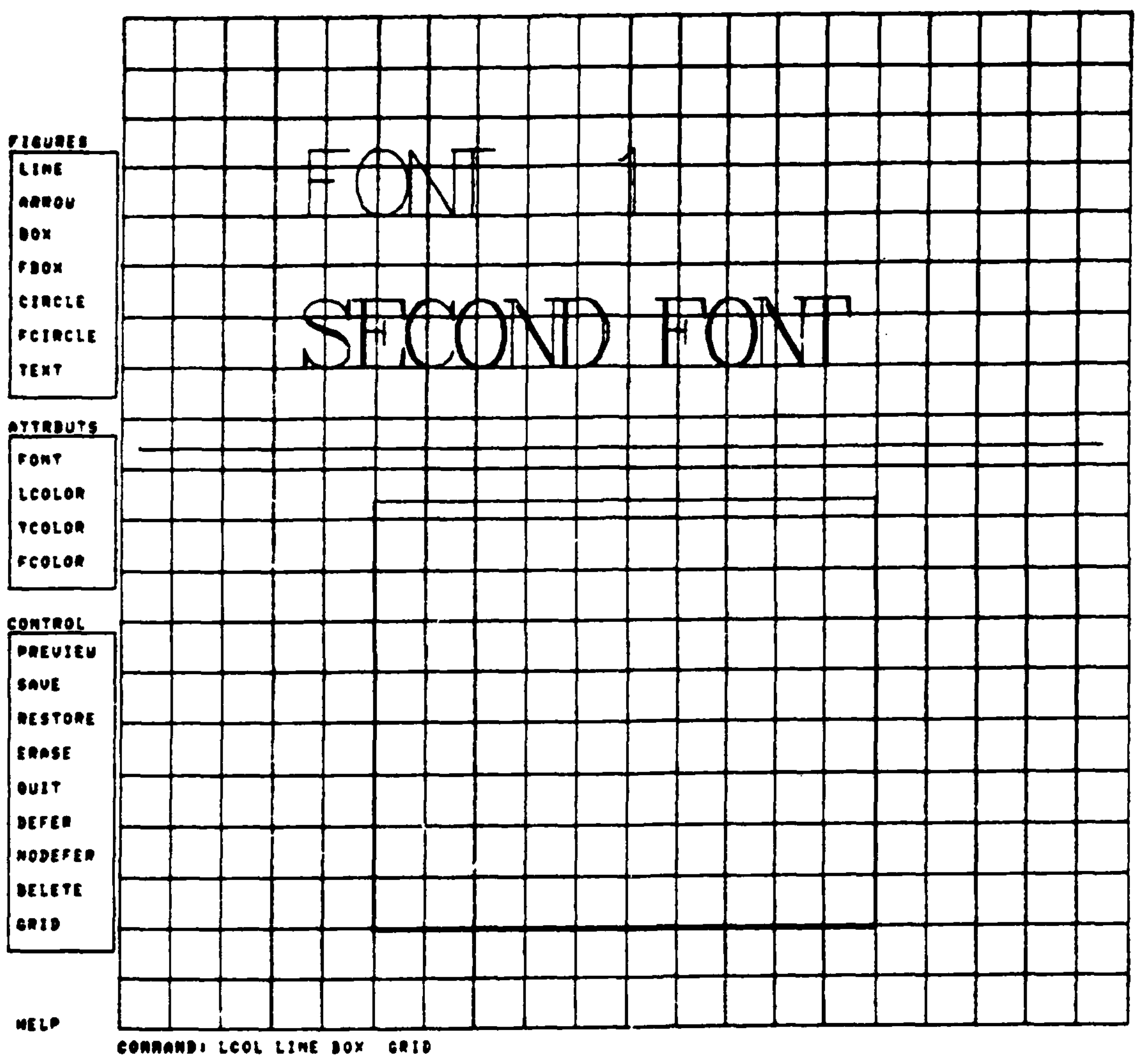
The second example shows the use of the line-color command, LCOL. For the Tektronix, the color attribute is ignored and the line is drawn as usual. However, if this picture is saved, then redrawn with the META program on the ISC color device, the line appears in the same place but with the correct color; in this case the color selected was 3, or green. After selecting the color, the next input was LINE for drawing the line itself. This time the graphics crosshairs must be positioned twice, once for each endpoint of the line. The third command was BOX; this command produces a box when the locations for the opposite corners of the box are selected with the crosshairs. The final command shown in the second example is GRID which has no immediate feedback. However, if PREVIEW, DELETE or ERASE is entered, when the new 'grid' is drawn, it appears as a collection of small crosses as shown in the third example. Note that PREVIEW redraws the current picture without the grid, prompts, and error messages, if any. PREVIEW should be entered periodically to 'clean up' the screen.

The third example shows the alternative grid style and another figure. The CIRCLE command produces the crosshairs, and the center and any point on the circumference can be selected. After selecting the second point the circle is immediately drawn. The remainder of the commands are described in the following text.



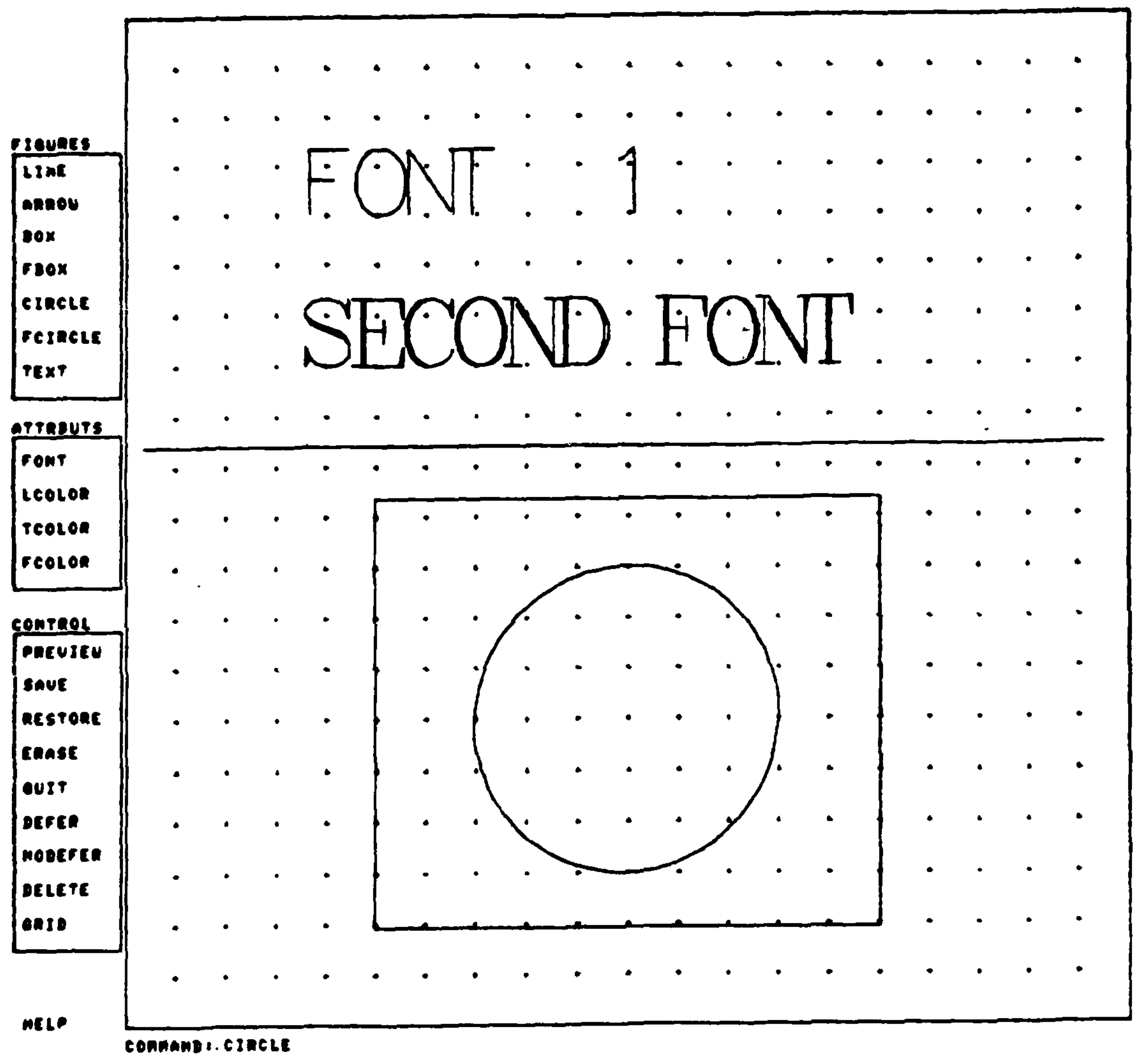
```
COMMAND: FONT TEX

FONT TYPE? (INTEGER 1-4): 3
```



```
COMMAND: LCOL LINE BOX  GRID

                              LINE INDEX?  COLORS 1..8: 3
```


FIGURES
LINE
ARROW
BOX
FBOX
CIRCLE
FCIRCLE
TEXT
ATTRBUTS
FONT
LCOLOR
TCOLOR
FCOLOR
CONTROL
PREVIEW
SAVE
RESTORE
ERASE
QUIT
DEFER
NODEFER
DELETE
GRID
HELP
FONT 1
SECOND FONT
COMMAND:.CIRCLE

Commands for INGRED

FIGURES

The figures represent the available objects or segments that can be drawn on the display under user control. The current repetoire includes:

(1) LINES with the current line color attribute

(2) an ARROWHEAD (in line color)

(3) a BOX of arbitrary dimensions (in line color)

(4) a filled box (FBOX) filled with the current color value for filling

(5) a filled circle (FCIRCLE) is NOT YET AVAILABLE.

(6) and TEXT with current font and text color attributes. Additional HELP information is available on each figure.

LINE

The LINE figure command produces the graphics crosshair cursor. The endpoints for the line should be selected by using the space bar. The first selection will be echoed by a small cross placed at the line endpoint. The crosshairs will reappear and the end point should be selected. The line will then be drawn on the screen. The LINE command may be abbreviated to LIN. The LINE command may be aborted when the crosshairs are on the screen by typing 'A' instead of using the space bar to select an endpoint.

BOX

The BOX figure command produces the graphics crosshair cursor. The opposite corners for the box should be selected by using the space bar. The first selection will be echoed by a small cross placed at the box corner. The crosshairs will reappear and the opposite corner should be selected. The box will then be drawn on the screen. The BOX command may be aborted when the crosshairs are on the screen by typing 'A' instead of using te space bar to select an endpoint.

CIRCLE

The CIRCLE figure command produces the graphics crosshair cursor. The center and any point on the circle should be selected by using the space bar. The center must be selected first and will be echoed by a cross. The graphics crosshairs will then reappear and the point on the circle should be selected. The circle will then be drawn on the screen. The CIRCLE command may be abbreviated to CIR. The CIRCLE command may be aborted when the crosshairs are on the screen by typing 'A' instead of using the space bar to select the center or point on the circle.

TEXT

The TEXT command produces the graphics crosshair cursor. The location of the lower left corner of the text string should be selected by using the thumbwheel cursors and entering the point by hitting the space bar. The block cursor will then appear at the selected point. The text may be entered from the keyboard. (RUB OUT is used to backspace and delete errors.) When the text is complete the return key is pressed to enter the string. The text will then be redrawn on the display. Note that if the font attribute is other than the Tektronix hardware font the input (echoed) text will be different from the final display text. The TEXT command may be abbreviated to TEX. The text command may be aborted when the crosshair cursor is displayed by typing 'A' instead of using the space bar to enter the text start point.

ATTRIBUTES

The attributes are used to control the quality of the figures displayed on the view surface. The attributes include the following:

1) The line color is specified by entering the LCOLOR command

2) The polygon fill color is specified by FCOLOR

3) The text color is specified by TCOLOR

4) The text (software) font style is specified by FONT

All of the attributes may be input by using the first three letters of the command. All of the attributes have further HELP available.

LCOLOR

The color attribute is used to specify the color of three primitive output types. These primitives are line, text and polygon fill color. The color attribute is an integer value in the range one to eight. The default attribute value for all primitives is white. Once an attribute is set it remains in effect until altered. The values for the color indices are given below.

ATTRIBUTES AND VALUES

BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5
YELLOW	6
MAGENTA	7
WHITE	8

FCOLOR

The color attribute is used to specify the color of three primitive output types. These primitives are line, text and polygon fill color. The color attribute is an integer value in the range one to eight. Th default attribute value for all primitives is white. Once an attribute is set it remains in effect until altered. The values for the color indices are given below.

ATTRIBUTES AND VALUES

BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5
YELLOW	6
MAGENTA	7
WHITE	8

TCOLOR

The color attribute is used to specify the color of three primitive output types. These primitives are line, text and polygon fill color. The color attribute is an integer value in the range one to eight. The default attribute value for all primitives is white. Once an attribute is set it remains in effect until altered. The values for the color indices are given below.

ATTRIBUTES AND VALUES

BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5
YELLOW	6
MAGENTA	7
WHITE	8

FONT

The font attribute determines the appearance of text displayed on the screen in response to the TEXT command. There are five font styles o available.

Ø ..... Tektronix hardware font (D)

1 ..... CORE software font — MODERN

2 ..... CORE software font — ROMAN

3 ..... CORE software font -- ROMAN Italics

4 ..... CORE software font — BLOCK

Only the Tekronix hadware font is proportionally spaced. To select a font style enter the FONT command and a number from Ø to 4.

CONTROL

The control instructions provide the user with commands for manipulating pictures generated by INGRED. The following functions may be performed:

(1) SAVE a picture to a file

(2) RESTORE a picture from a file

(3) ERASE the current display file

(4) alter the GRID style

(5) DELETE selected figures

(6) DEFER the deletions

(7) QUIT the current interactive session.

All of the control commands may be abbreviated to the first three letters. Additional HELP information is available on individual commands.

PREVIEW

Preview allows the current picture to be displayed without the positioning grid and command prompts. This instruction also clears the screen of extraneous input and error messages and should be used periodically to clean up the screen area. The space bar will return the user to the editing mode.

SAVE

Save allows the user to place the current picture on disk so that it may be retrieved at a later time for further editing. The SAVE prompt requests a file name; only the file name is necessary, the default file type is supplied by the editor.

RESTORE

RESTORE allows the user to retrieve a previously saved picture file from the disk for further editing or display. The restore command deletes the current picture information from the display file before reading in the new picture. The RESTORE prompt asks for a file name; only the name is necessary, the file type is supplied by the editor.

ERASE

ERASE deletes the current segments from the display file and the current picture is lost.

DELETE

DELETE produces the graphics crosshairs on the display surface. The intersection of the crosshairs should be positioned with the thumbwheel within the figure to be deleted (or in the case of text, near the start of the text line). When the cursor is correctly positioned the space bar may be used to pick the figure or text to be deleted. The editor will prompt by

(1) redrawing the selected figure

(2) and waiting for yes/no confirmation. If 'NO' is chosen in response to the prompt then the crosshairs will reappear and may be repositioned. To abort deletion, an 'A' may be hit instead of the space bar and the editor will return to the command level.

DEFER

DEFER and NODEFER control the updating of the screen in response to deletions. If many deletions are to be accomplished then the DEFER mode prevents the screen from erasing and redrawing the segements until the last deletion occurs. DEFER and NODEFER may be set at any time. The default is NODEFER. (-- not presently implemented --)

NODEFER (D)

DEFER and NODEFER control the updating of the screen in response to deletions. If many deletions are to be accomplished then the DEFER mode prevents the screen from erasing and redrawing the segments until the last deletion occurs. DEFER and NODEFER may be set at any time. (-- not presently implemented --)

QUIT

QUIT terminates the interactive editor. The information in the display file is lost unless previously saved.

GRID

The GRID command changes the rectangular grid on the display surface to a set of points that mark the intersections of the (default) grid. The command GRID is toggled when the command is entered. The default is the rectilinear grid; upon the first invocation the grid will be replaced by the points. The second invocation will return the grid to the default grid style. [NOTE: The change is not instantaneous, the most recently selected grid will appear after a deletion, erase or a preview command.]

VITA

Kevin Wilson Rose was born February 25, 1955 in Ossining, New York. He graduated from Downingtown Senior High School, Downingtown, Pennsylvania in 1973. He then studied electrical engineering at Cornell University, Ithaca, New York from which he received the degree of Bachelor of Science in May, 1977. Upon graduation he was commissioned an officer in the United States Air Force. From October, 1977 to May 1980 he was assigned to the Air Force Satellite Control Facility, Sunnyvale, California as a development planning engineer. From May 1980 to May 1981 he was an Operation's Director for the Satellite Data System's Program Office at the Satellite Control Facility in the directorate of operations. He entered the Air Force Institute of Technology's Masters degree program in June 1981.

Permanent Address: 227 McIlvain Drive

Downingtown, Pennsylvania 19335

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER AFIT/GE/EE/82D-58	2. GOVT ACCESSION NO. AD-A124694	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DEVELOPMENT OF AN INTERACTIVE COMPUTER GRAPHICS SOFTWARE SYSTEM LIBRARY AND GRAPHICS TOOL		5. TYPE OF REPORT & PERIOD COVERED MS Thesis
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kevin W. Rose Capt, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December, 1982
		13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

Approved for public release: IAW AFR 190-17

[illegible]

LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

4 JAN 1983

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Graphics, Software Tools
Interactive Graphics
Core Standard

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The requirements for a general purpose, interactive graphics software system were investigated and led to the development of a graphics library of subroutines which can be used as a graphics research tool or for applications development. The design objective for software environments and software tools established the strategy for developing the graphics software library. It was determined that the ACM Core standard graphics specification

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

satisfied many of the requirements. An output-only implementation
of the Core by George Washington University was extended to in-
clude a device driver for a color display device and the
synchronous input primitives. The graphics system was used to
develop an application program to demonstrate the design of an
effective user-tool interface. The program is an interactive
graphics editor that allows generation of two-dimensional
pictures and includes capabilities for picture storage and re-
trieval, segment addition and deletion, attribute control, error
handling and on-line help. The graphics software library
implemented to the specification of the Core standard proposal
provides an effective tool for interactive computer graphics
education, research, and applications development.